Title: Pipelined SIMD Multimedia Unit Design Report

Saphal Baral
Undergraduate Student, Computer Engineering
Stony Brook University

Harry Lin
Undergraduate Student, Computer Engineering
Stony Brook University

Course: ESE 345 - Computer Architecture
Instructor: Mikhail Dorojevets

Date: November 30, 2025

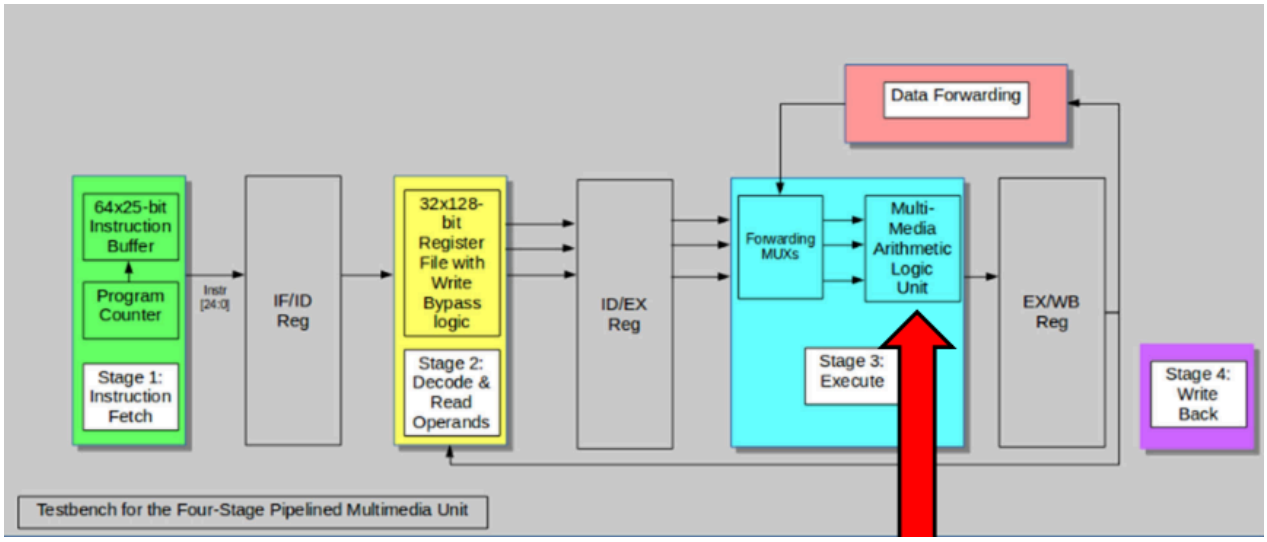
Table of Contents

Objective

The goal of the project was to implement a fully functional four-stage pipelined multimedia processor in VHDL, capable of executing 25-bit instructions across a 128-bit register architecture. The processor supports three instruction formats: LI, R3, and R4 and implements a complete pipeline consisting of Instruction Fetch, Decode/Register Read, Execute (MALU), and Write-Back stages. Here were some key design goals:

- Correct operand decoding.
- Hazard-free execution using data forwarding.
- Verification of behavioral and structural architecture through simulation and waveform analysis.

A custom assembler was developed to translate human-readable assembly into binary machine format and the resulting code was loaded into the Instruction Buffer for execution. The project demonstrates pipeline capabilities through correct hazard handling and end-to-end instruction flow through all pipeline stages.



Block Diagram. Block diagram of the pipelined multimedia unit design.

Design Procedure

1. MALU Development

The project began by designing and validating the MALU (Multimedia Arithmetic Logic Unit) before constructing the actual pipeline. This allowed all instruction behaviors to be verified independently of pipeline hazards and decode logic. Here were some key design goals:

- Implementing LI, R3, and R4 instruction formats.
- Produce 128-bit saturated results based on opcode.
- Support LI immediate insertion using Load Index.

2. Pipeline Architecture Construction

With the MALU working, the remaining system was built around it, forming a four-stage pipeline:

a. Instruction Fetch (IF)

- Program Counter increments sequentially.
- Instruction Buffer holds 64 x 25-bit instructions.
- IF/ID register latches instructions at each rising edge.

b. Instruction Decode + Register File (ID)

- 25-bit instruction is assigned an instruction format and decomposed into rd, rs1, rs2, rs3, opcode, immediate, and load index as according to the instruction format.
- Register File implemented with three asynchronous read ports and one synchronous write port.
- Write-Bypass logic added so a register read immediately following a write returns the updated value rather than stale data.

c. Execute (EX)

- ID/EX register transfers the full decoded instruction state.
- Forward MUXs inserted before MALU inputs to resolve any RAW hazards.
- MALU reused from design procedure step 1 and integrated into the pipeline path.

d. Write-Back (WB)

- EX/WB register latches result with the rd index.
- Data is written to the Register File if reg_write = '1'.

3. Hazard Detection and Data Forwarding

Without any protection, the CPU would attempt to read a register before the previous instruction finished writing to it, creating a RAW (Read After Write) hazard. Rather than stalling the pipeline, the processor must forward results directly from later pipeline stages, allowing execution to continue every cycle.

4. Assembly Encoding

When the pipeline and data forwarding was complete, a C++ assembler was written to convert the symbolic instructions into 25-bit binary form. A program.txt file was loaded into the Instruction Buffer during simulation.

5. Testbench Simulation

The final top-level module connected all pipeline components structurally. A VHDL testbench was written to do the following:

- Load machine code program into the Instruction Buffer.
- Run clock cycling and deassert reset.
- Observe the register updates and MALU results in waveform.
- Verify the correctness using forwarding-dependent programs.

Load Immediate Instruction Format

Figure 1. The instruction format for load immediate.

Bit Range	Field Name	Description
24	Opcode (0)	Identifies the instruction as a load immediate operation.
23-21	Load Index	A field used to differentiate between different load variants.
20-5	16-bit Immediate	Contains the constant value to be loaded into rd.
4-0	rd	Destination register.

Table 1. Table explaining the bits of the load immediate instruction format.

Instruction	Description
li: load immediate	Loads a 16-bit value from bits [20:5] into the 16-bit segment of the 128-bit register rd specified by the Load Index [23:21]. Other segments of rd remain unchanged. Since the instruction reads, modifies, and writes back rd, it acts as both the source and destination register.

Table 2. Table explaining all instructions in load immediate instruction format.

li: load immediate

```
-- Load Immediate
when "00" | "01" =>
load_index := to_integer(unsigned(instruction_format(23 downto 21)));       -- Extract the load index
immediate := unsigned(instruction_format(20 downto 5));                  -- Extract the immediate value
output := rd;                                                   -- Reads register in which the source = destination
output(load_index*16 + 15 downto load_index*16) := std_logic_vector(immediate); -- Place immediate value in correct load index field
rd <= output;                                                   -- Output new result
```

Figure 2. Source code for load immediate instruction.

```
report "=== STARTING LOAD IMMEDIATE TESTBENCH ===";
-- TEST 1
-- AND rs1 and rs2 just to have a rd that is not 'U'
rs1 <= x"11111111111111111111111111111111";
rs2 <= x"11111111111111111111111111111111";
instruction_format <= "11" & "00001011" & "000000000000000";
wait for period;

-- Format ID: "00", Load Index = "011", Immediate = x"F00D", rd = "00000"
instruction_format <= "0" & "011" & x"F00D" & "00000";
wait for period;

-- TEST 2
-- AND rs1 and rs2 just to have a rd that is not 'U'
rs1 <= x"11111111111111111111111111111111";
rs2 <= x"11111111111111111111111111111111";
instruction_format <= "11" & "00001011" & "000000000000000";
wait for period;

-- Format ID: "01", Load Index = "101", Immediate = x"BEEF", rd = "01010"
instruction_format <= "0" & "101" & x"BEEF" & "01010";
wait for period;

-- TEST 3
-- AND rs1 and rs2 just to have a rd that is not 'U'
rs1 <= x"11111111111111111111111111111111";
rs2 <= x"11111111111111111111111111111111";
instruction_format <= "11" & "00001011" & "000000000000000";
wait for period;

-- Format ID: "01", Load Index = "111", Immediate = x"DEAD", rd = "11011"
instruction_format <= "0" & "111" & x"DEAD" & "11011";
wait for period;

report "All load immediate tests complete!";
```

Figure 3. Testbench code for load immediate instruction.

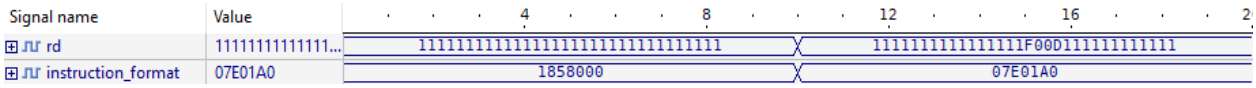


Figure 4. Waveform for test case 1 of load immediate instruction.

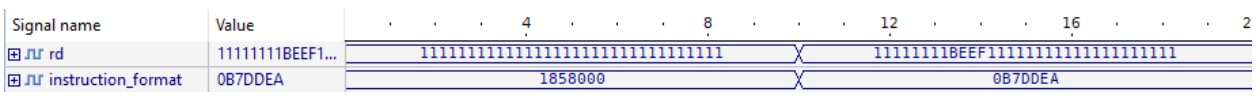


Figure 5. Waveform for test case 2 of load immediate instruction.

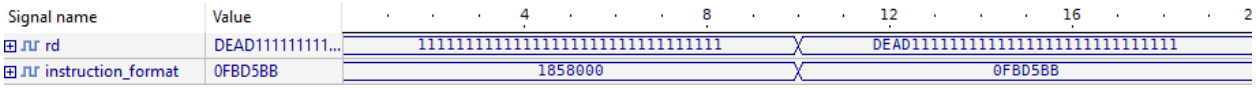


Figure 6. Waveform for test case 3 of load immediate instruction.

Results:

We used the AND instruction in the beginning to begin the test cases with a rd value that is not uninitialized. Thus, from 0-10ns, that is the result of the AND instruction. 10-20ns displays placing the immediate value in the correct load index. All test cases were successful and produced expected results.

R4 Instruction Format

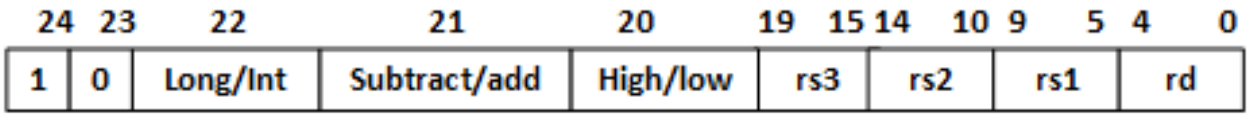


Figure 7. The instruction format for R4 instructions.

Bit Range	Field Name	Description
24	Opcode (1)	First bit to identify the instruction format.
23	Opcode (0)	The second bit is used to verify this is the R4 instruction format ('10').
22	Long/Int	Selects between 64-bit long or 32-bit integer mode.
21	Subtract/Add	Determines whether the operation performs addition (0) or subtraction (1).
20	High/Low	Selects the upper or lower half of the 128-bit register for the operation.
19-15	rs3	Third source register.
14-10	rs2	Second source register.
9-5	rs1	First source register.
4-0	rd	Destination register.

Table 3. Table explaining the bits of the R4 instruction format.

Size (Num Bits)	Min	Max
Long (64)	-2^63	+2^63 - 1
Int (32)	-2^31	+2^31 - 1

Table 4. Table displaying the minimum and maximum values used for saturated rounding. Instead of over/underflow wrapping, the above values are used.

LI/SA/HL [22:20]	Instruction	Description
000	Signed Integer Multiply - Add Low with Saturation (SIMALWS)	Multiplies the low 16-bit segments of each 32-bit field in registers rs3 and rs2, adds the resulting 32-bit products to the corresponding 32-bit fields of rs1, and stores the saturated result in rd.
001	Signed Integer Multiply - Add High with Saturation (SIMAHWS)	Multiplies the high 16-bit segments of each 32-bit field in rs3 and rs2, adds the resulting 32-bit products to the 32-bit fields of rs1, and writes the saturated result to rd.
010	Signed Integer Multiply - Subtract Low with Saturation (SIMSLWS)	Multiplies the low 16-bit segments of each 32-bit field in rs3 and rs2, subtracts the resulting 32-bit products from the corresponding 32-bit fields of rs1, and stores the saturated result in rd.
011	Signed Integer Multiply - Subtract High with Saturation (SIMSHWS)	Multiplies the high 16-bit segments of each 32-bit field in rs3 and rs2, subtracts the resulting 32-bit products from the 32-bit fields of rs1, and writes the saturated result to rd.
100	Signed Long Integer Multiply - Add Low with Saturation (SLIMALWS)	Multiplies the low 32-bit segments of each 64-bit field in rs3 and rs2, adds the resulting 64-bit products to the 64-bit fields of rs1, and stores the saturated result in rd.
101	Signed Long Integer Multiply - Add High with Saturation (SLIMAHWS)	Multiplies the high 32-bit segments of each 64-bit field in rs3 and rs2, adds the resulting 64-bit products to the 64-bit fields of rs1, and

		writes the saturated result to rd.
110	Signed Long Integer Multiply - Subtract Low with Saturation (SLIMSLWS)	Multiplies the low 32-bit segments of each 64-bit field in rs3 and rs2, subtracts the resulting 64-bit products from the 64-bit fields of rs1, and stores the saturated result in rd.
111	Signed Long Integer Multiply - Subtract High with Saturation (SLIMSHWS)	Multiplies the high 32-bit segments of each 64-bit field in rs3 and rs2, subtracts the resulting 64-bit products from the 64-bit fields of rs1, and writes the saturated result to rd.

Table 5. Table explaining all instructions in R4 instruction format.

Signed Integer Multiply - Add Low with Saturation (SIMALWS)

```
-- Signed Integer Multiply - Add Low with Saturation
when "000" =>
-- Word 0 [31:0]
product_int_32 := signed(rs2(15 downto 0)) * signed(rs3(15 downto 0));
sum_int_32 := resize(signed(rs1(31 downto 0)), 33) + resize(product_int_32, 33);
if sum_int_32 > resize(signed_int_32_max, 33) then
    output(31 downto 0) := STD_LOGIC_VECTOR(signed_int_32_max);
elsif sum_int_32 < resize(signed_int_32_min, 33) then
    output(31 downto 0) := STD_LOGIC_VECTOR(signed_int_32_min);
else
    output(31 downto 0) := STD_LOGIC_VECTOR(sum_int_32(31 downto 0));
end if;

-- Word 1 [63:32]
product_int_32 := signed(rs2(47 downto 32)) * signed(rs3(47 downto 32));
sum_int_32 := resize(signed(rs1(63 downto 32)), 33) + resize(product_int_32, 33);
if sum_int_32 > resize(signed_int_32_max, 33) then
    output(63 downto 32) := STD_LOGIC_VECTOR(signed_int_32_max);
elsif sum_int_32 < resize(signed_int_32_min, 33) then
    output(63 downto 32) := STD_LOGIC_VECTOR(signed_int_32_min);
else
    output(63 downto 32) := STD_LOGIC_VECTOR(sum_int_32(31 downto 0));
end if;

-- Word 2 [95:64]
product_int_32 := signed(rs2(79 downto 64)) * signed(rs3(79 downto 64));
sum_int_32 := resize(signed(rs1(95 downto 64)), 33) + resize(product_int_32, 33);
if sum_int_32 > resize(signed_int_32_max, 33) then
    output(95 downto 64) := STD_LOGIC_VECTOR(signed_int_32_max);
elsif sum_int_32 < resize(signed_int_32_min, 33) then
    output(95 downto 64) := STD_LOGIC_VECTOR(signed_int_32_min);
else
    output(95 downto 64) := STD_LOGIC_VECTOR(sum_int_32(31 downto 0));
end if;

-- Word 3 [127:96]
product_int_32 := signed(rs2(111 downto 96)) * signed(rs3(111 downto 96));
sum_int_32 := resize(signed(rs1(127 downto 96)), 33) + resize(product_int_32, 33);
if sum_int_32 > resize(signed_int_32_max, 33) then
    output(127 downto 96) := STD_LOGIC_VECTOR(signed_int_32_max);
elsif sum_int_32 < resize(signed_int_32_min, 33) then
    output(127 downto 96) := STD_LOGIC_VECTOR(signed_int_32_min);
else
    output(127 downto 96) := STD_LOGIC_VECTOR(sum_int_32(31 downto 0));
end if;
```

Figure 8. Source code for SIMALWS instruction.

```
-- SIMALWS --
report "=== STARTING SIGNED INTEGER MULTIPLY - ADD LOW WITH SATURATION TESTBENCH ===";

-- TEST 1 (no saturation)
rs1 <= x"00000001000000020000000300000004";
rs2 <= x"00000002000000030000000400000005";
rs3 <= x"00000003000000040000000500000006";
instruction_format <= "10" & "000" & "0000000000000000000";
wait for period;

-- TEST 2 (+ saturation)
rs1 <= x"7FFFFFFF7FFFFFFF7FFFFFFF7FFFFFFF";
rs2 <= x"00008000000080000000800000008000";
rs3 <= x"0000FFFF0000FFFF0000FFFF0000FFFF";
instruction_format <= "10" & "000" & "0000000000000000000";
wait for period;

-- TEST 3 (- saturation)
rs1 <= x"80000000800000008000000080000000";
rs2 <= x"00008000000080000000800000008000";
rs3 <= x"00000001000000010000000100000001";
instruction_format <= "10" & "000" & "0000000000000000000";
wait for period;

-- TEST 4 (mixed signs)
rs1 <= x"00000001000000020000000300000004";
rs2 <= x"0000FFFF0000FFFE0000FFFD0000FFFC";
rs3 <= x"00000001000000020000000300000004";
instruction_format <= "10" & "000" & "0000000000000000000";
wait for period;

report "All signed integer multiply - add low with saturation tests complete!";
```

Figure 9. Testbench code for SIMALWS instruction.

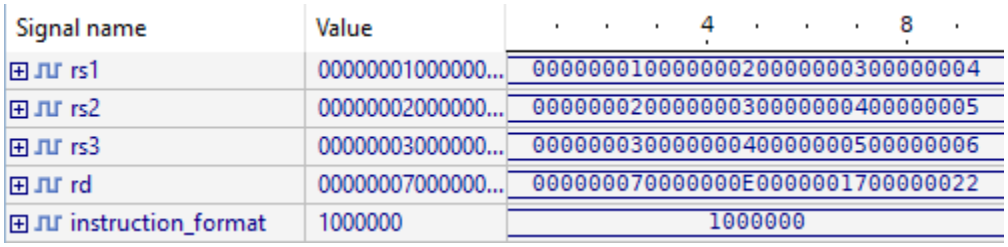


Figure 10. Waveform for test case 1 of SIMALWS instruction.

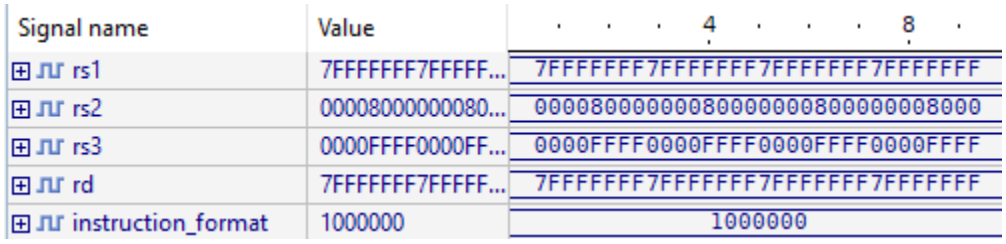


Figure 11. Waveform for test case 2 of SIMALWS instruction.

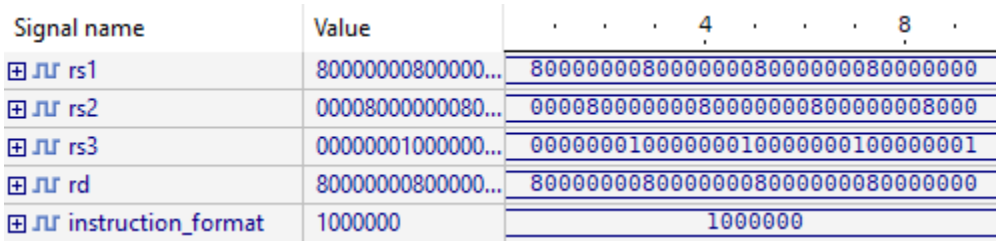


Figure 12. Waveform for test case 3 of SIMALWS instruction.

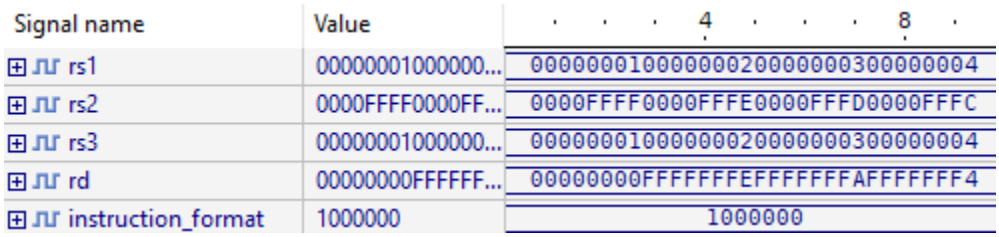


Figure 13. Waveform for test case 4 of SIMALWS instruction.

Results:

The test cases were designed to test no saturation, positive saturation, negative saturation, and registers with mixed signs. All test cases were successful and produced expected results.

Signed Integer Multiply - Add High with Saturation (SIMAHWS)

```
-- Signed Integer Multiply - Add High with Saturation
when "001" =>
-- Word 0 [31:0]
product_int_32 := signed(rs2(31 downto 16)) * signed(rs3(31 downto 16));
sum_int_32 := resize(signed(rs1(31 downto 0)), 33) + resize(product_int_32, 33);
if sum_int_32 > resize(signed_int_32_max, 33) then
    output(31 downto 0) := STD_LOGIC_VECTOR(signed_int_32_max);
elsif sum_int_32 < resize(signed_int_32_min, 33) then
    output(31 downto 0) := STD_LOGIC_VECTOR(signed_int_32_min);
else
    output(31 downto 0) := STD_LOGIC_VECTOR(sum_int_32(31 downto 0));
end if;

-- Word 1 [63:32]
product_int_32 := signed(rs2(63 downto 48)) * signed(rs3(63 downto 48));
sum_int_32 := resize(signed(rs1(63 downto 32)), 33) + resize(product_int_32, 33);
if sum_int_32 > resize(signed_int_32_max, 33) then
    output(63 downto 32) := STD_LOGIC_VECTOR(signed_int_32_max);
elsif sum_int_32 < resize(signed_int_32_min, 33) then
    output(63 downto 32) := STD_LOGIC_VECTOR(signed_int_32_min);
else
    output(63 downto 32) := STD_LOGIC_VECTOR(sum_int_32(31 downto 0));
end if;

-- Word 2 [95:64]
product_int_32 := signed(rs2(95 downto 80)) * signed(rs3(95 downto 80));
sum_int_32 := resize(signed(rs1(95 downto 64)), 33) + resize(product_int_32, 33);
if sum_int_32 > resize(signed_int_32_max, 33) then
    output(95 downto 64) := STD_LOGIC_VECTOR(signed_int_32_max);
elsif sum_int_32 < resize(signed_int_32_min, 33) then
    output(95 downto 64) := STD_LOGIC_VECTOR(signed_int_32_min);
else
    output(95 downto 64) := STD_LOGIC_VECTOR(sum_int_32(31 downto 0));
end if;

-- Word 3 [127:96]
product_int_32 := signed(rs2(127 downto 112)) * signed(rs3(127 downto 112));
sum_int_32 := resize(signed(rs1(127 downto 96)), 33) + resize(product_int_32, 33);
if sum_int_32 > resize(signed_int_32_max, 33) then
    output(127 downto 96) := STD_LOGIC_VECTOR(signed_int_32_max);
elsif sum_int_32 < resize(signed_int_32_min, 33) then
    output(127 downto 96) := STD_LOGIC_VECTOR(signed_int_32_min);
else
    output(127 downto 96) := STD_LOGIC_VECTOR(sum_int_32(31 downto 0));
end if;
```

Figure 14. Source code for SIMAHWS instruction.

```
-- SIMAHWS --
report "=== STARTING SIGNED INTEGER MULTIPLY - ADD HIGH WITH SATURATION TESTBENCH ===";

-- TEST 1 (no saturation)
rs1 <= x"00000001000000020000000300000004";
rs2 <= x"00000002000000030000000400000005";
rs3 <= x"00000003000000040000000500000006";
instruction_format <= "10" & "001" & "0000100000010000100";
wait for period;

-- TEST 2 (+ saturation)
rs1 <= x"7FFFFFFF7FFFFFFF7FFFFFFF7FFFFFFF";
rs2 <= x"00008000000080000000800000008000";
rs3 <= x"0000FFFF0000FFFF0000FFFF0000FFFF";
instruction_format <= "10" & "001" & "0000000000000000000";
wait for period;

-- TEST 3 (- saturation)
rs1 <= x"80000000800000008000000080000000";
rs2 <= x"00008000000080000000800000008000";
rs3 <= x"00000001000000010000000100000001";
instruction_format <= "10" & "001" & "0000000100001000000";
wait for period;

-- TEST 4 (mixed signs)
rs1 <= x"00000001000000020000000300000004";
rs2 <= x"7FFF80007FFF80007FFF80007FFF8000";
rs3 <= x"00010001000100010001000100010001";
instruction_format <= "10" & "001" & "0000000000000000000";
wait for period;

report "All signed integer multiply - add high with saturation tests complete!";
```

Figure 15. Testbench code for SIMAHWS instruction.

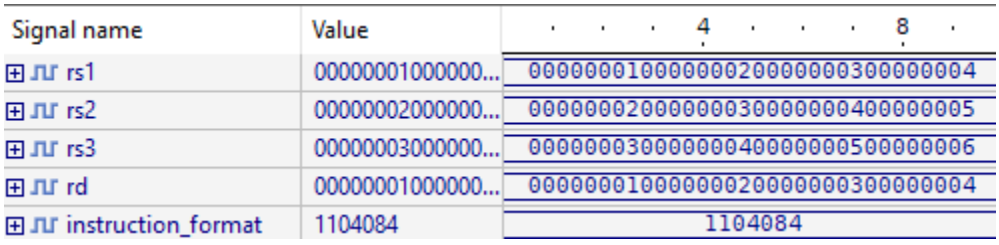


Figure 16. Waveform for test case 1 of SIMAHWS instruction.

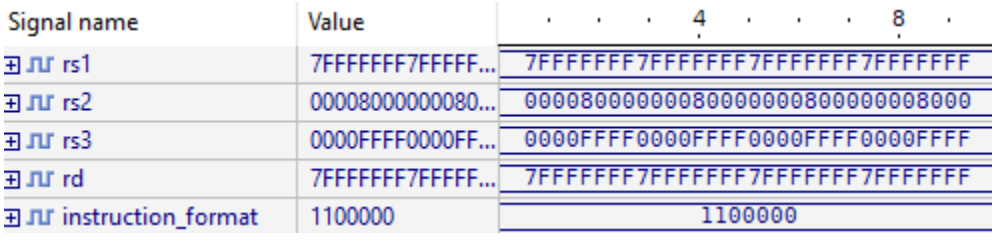


Figure 17. Waveform for test case 2 of SIMAHWS instruction.

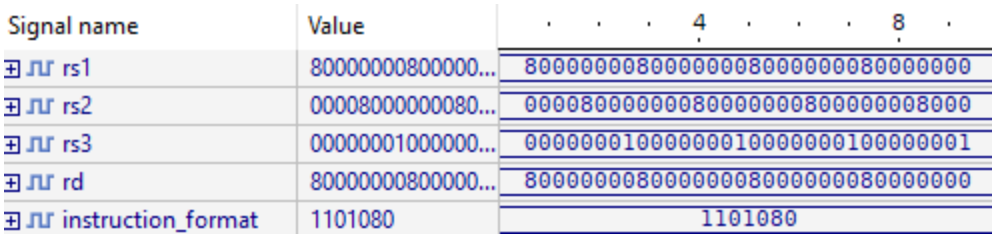


Figure 18. Waveform for test case 3 of SIMAHWS instruction.

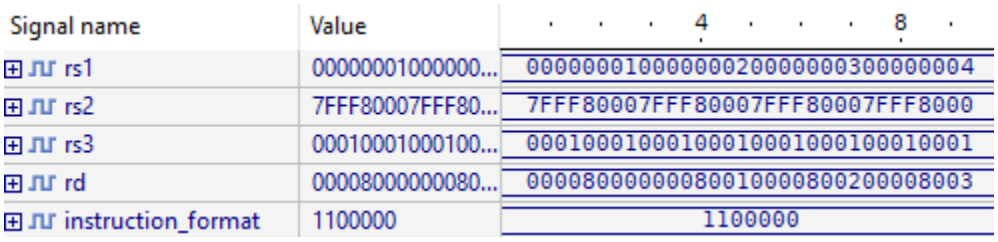


Figure 19. Waveform for test case 4 of SIMAHWS instruction.

Results:

The test cases were designed to test no saturation, positive saturation, negative saturation, and registers with mixed signs. All test cases were successful and produced expected results.

Signed Integer Multiply - Subtract Low with Saturation (SIMSLWS)

```
-- Signed Integer Multiply - Subtract Low with Saturation
when "010" =>
-- Word 0 [31:0]
product_int_32 := signed(rs2(15 downto 0)) * signed(rs3(15 downto 0));
diff_int_32 := resize(signed(rs1(31 downto 0)), 33) - resize(product_int_32, 33);
if diff_int_32 > resize(signed_int_32_max, 33) then
    output(31 downto 0) := STD_LOGIC_VECTOR(signed_int_32_max);
elsif diff_int_32 < resize(signed_int_32_min, 33) then
    output(31 downto 0) := STD_LOGIC_VECTOR(signed_int_32_min);
else
    output(31 downto 0) := STD_LOGIC_VECTOR(diff_int_32(31 downto 0));
end if;

-- Word 1 [63:32]
product_int_32 := signed(rs2(47 downto 32)) * signed(rs3(47 downto 32));
diff_int_32 := resize(signed(rs1(63 downto 32)), 33) - resize(product_int_32, 33);
if diff_int_32 > resize(signed_int_32_max, 33) then
    output(63 downto 32) := STD_LOGIC_VECTOR(signed_int_32_max);
elsif diff_int_32 < resize(signed_int_32_min, 33) then
    output(63 downto 32) := STD_LOGIC_VECTOR(signed_int_32_min);
else
    output(63 downto 32) := STD_LOGIC_VECTOR(diff_int_32(31 downto 0));
end if;

-- Word 2 [95:64]
product_int_32 := signed(rs2(79 downto 64)) * signed(rs3(79 downto 64));
diff_int_32 := resize(signed(rs1(95 downto 64)), 33) - resize(product_int_32, 33);
if diff_int_32 > resize(signed_int_32_max, 33) then
    output(95 downto 64) := STD_LOGIC_VECTOR(signed_int_32_max);
elsif diff_int_32 < resize(signed_int_32_min, 33) then
    output(95 downto 64) := STD_LOGIC_VECTOR(signed_int_32_min);
else
    output(95 downto 64) := STD_LOGIC_VECTOR(diff_int_32(31 downto 0));
end if;

-- Word 3 [127:96]
product_int_32 := signed(rs2(111 downto 96)) * signed(rs3(111 downto 96));
diff_int_32 := resize(signed(rs1(127 downto 96)), 33) - resize(product_int_32, 33);
if diff_int_32 > resize(signed_int_32_max, 33) then
    output(127 downto 96) := STD_LOGIC_VECTOR(signed_int_32_max);
elsif diff_int_32 < resize(signed_int_32_min, 33) then
    output(127 downto 96) := STD_LOGIC_VECTOR(signed_int_32_min);
else
    output(127 downto 96) := STD_LOGIC_VECTOR(diff_int_32(31 downto 0));
end if;
```

Figure 20. Source code for SIMSLWS instruction.

```
-- SIMSLWS --
report "=== STARTING SIGNED INTEGER MULTIPLY - SUBTRACT LOW WITH SATURATION TESTBENCH ===";

-- TEST 1 (no saturation)
rs1 <= x"00000001000000020000000300000004";
rs2 <= x"00000002000000030000000400000005";
rs3 <= x"00000003000000040000000500000006";
instruction_format <= "10" & "010" & "00000000000000000000";
wait for period;

-- TEST 2 (+ saturation)
rs1 <= x"7FFFFFFF7FFFFFFF7FFFFFFF7FFFFFFF";
rs2 <= x"00008000000080000000800000008000";
rs3 <= x"00000001000000010000000100000001";
instruction_format <= "10" & "010" & "00001000000001000100";
wait for period;

-- TEST 3 (- saturation)
rs1 <= x"80000000800000008000000080000000";
rs2 <= x"00000002000000020000000200000002";
rs3 <= x"00007FFF00007FFF00007FFF00007FFF";
instruction_format <= "10" & "010" & "00000000010000101100";
wait for period;

-- TEST 4 (mixed signs)
rs1 <= x"00000001000000020000000300000004";
rs2 <= x"0000FFFF0000FFFE0000FFFD0000FFFC";
rs3 <= x"00000001000000020000000300000004";
instruction_format <= "10" & "010" & "00001001000000001000";
wait for period;

report "All signed integer multiply - subtract low with saturation tests complete!";
```

Figure 21. Testbench code for SIMSLWS instruction.

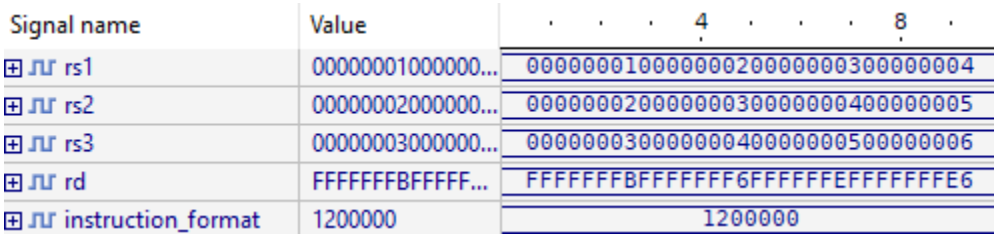


Figure 22. Waveform for test case 1 of SIMSLWS instruction.

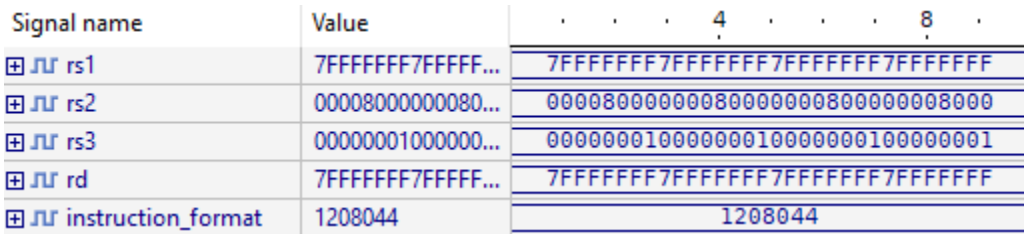


Figure 23. Waveform for test case 2 of SIMSLWS instruction.

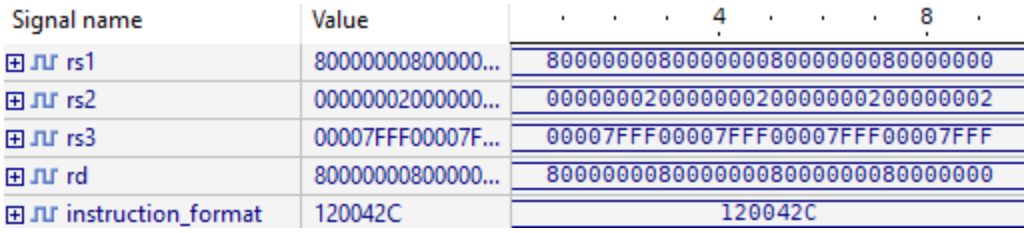


Figure 24. Waveform for test case 3 of SIMSLWS instruction.

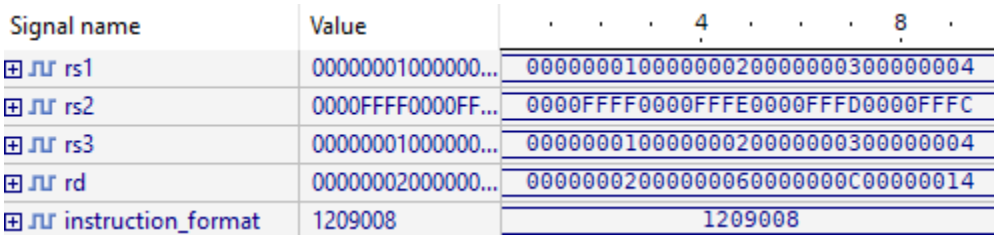


Figure 25. Waveform for test case 4 of SIMSLWS instruction.

Results:

The test cases were designed to test no saturation, positive saturation, negative saturation, and registers with mixed signs. All test cases were successful and produced expected results.

Signed Integer Multiply - Subtract High with Saturation (SIMSHWS)

```
-- Signed Integer Multiply - Subtract High with Saturation
when "011" =>
-- Word 0 [31:0]
product_int_32 := signed(rs2(31 downto 16)) * signed(rs3(31 downto 16));
diff_int_32 := resize(signed(rs1(31 downto 0)), 33) - resize(product_int_32, 33);
if diff_int_32 > resize(signed_int_32_max, 33) then
    output(31 downto 0) := STD_LOGIC_VECTOR(signed_int_32_max);
elsif diff_int_32 < resize(signed_int_32_min, 33) then
    output(31 downto 0) := STD_LOGIC_VECTOR(signed_int_32_min);
else
    output(31 downto 0) := STD_LOGIC_VECTOR(diff_int_32(31 downto 0));
end if;

-- Word 1 [63:32]
product_int_32 := signed(rs2(63 downto 48)) * signed(rs3(63 downto 48));
diff_int_32 := resize(signed(rs1(63 downto 32)), 33) - resize(product_int_32, 33);
if diff_int_32 > resize(signed_int_32_max, 33) then
    output(63 downto 32) := STD_LOGIC_VECTOR(signed_int_32_max);
elsif diff_int_32 < resize(signed_int_32_min, 33) then
    output(63 downto 32) := STD_LOGIC_VECTOR(signed_int_32_min);
else
    output(63 downto 32) := STD_LOGIC_VECTOR(diff_int_32(31 downto 0));
end if;

-- Word 2 [95:64]
product_int_32 := signed(rs2(95 downto 80)) * signed(rs3(95 downto 80));
diff_int_32 := resize(signed(rs1(95 downto 64)), 33) - resize(product_int_32, 33);
if diff_int_32 > resize(signed_int_32_max, 33) then
    output(95 downto 64) := STD_LOGIC_VECTOR(signed_int_32_max);
elsif diff_int_32 < resize(signed_int_32_min, 33) then
    output(95 downto 64) := STD_LOGIC_VECTOR(signed_int_32_min);
else
    output(95 downto 64) := STD_LOGIC_VECTOR(diff_int_32(31 downto 0));
end if;

-- Word 3 [127:96]
product_int_32 := signed(rs2(127 downto 112)) * signed(rs3(127 downto 112));
diff_int_32 := resize(signed(rs1(127 downto 96)), 33) - resize(product_int_32, 33);
if diff_int_32 > resize(signed_int_32_max, 33) then
    output(127 downto 96) := STD_LOGIC_VECTOR(signed_int_32_max);
elsif diff_int_32 < resize(signed_int_32_min, 33) then
    output(127 downto 96) := STD_LOGIC_VECTOR(signed_int_32_min);
else
    output(127 downto 96) := STD_LOGIC_VECTOR(diff_int_32(31 downto 0));
end if;
```

Figure 26. Source code for SIMSHWS instruction.

```
-- SIMSHWS --
report "=== STARTING SIGNED INTEGER MULTIPLY - SUBTRACT HIGH WITH SATURATION TESTBENCH ===";

-- TEST 1 (no saturation)
rs1 <= x"00000001000000020000000300000004";
rs2 <= x"00020003000400050006000700080009";
rs3 <= x"00010002000300040005000600070008";
instruction_format <= "10" & "011" & "00000000000000000000";
wait for period;

-- TEST 2 (+ saturation)
rs1 <= x"7FFFFFFF7FFFFFFF7FFFFFFF7FFFFFFF";
rs2 <= x"7FFF7FFF7FFF7FFF7FFF7FFF7FFF7FFF";
rs3 <= x"FFFF7FFF7FFF7FFF7FFF7FFF7FFF7FFF";
instruction_format <= "10" & "011" & "00000000010000000010";
wait for period;

-- TEST 3 (- saturation)
rs1 <= x"80000000800000008000000080000000";
rs2 <= x"7FFF7FFF7FFF7FFF7FFF7FFF7FFF7FFF";
rs3 <= x"7FFF7FFF7FFF7FFF7FFF7FFF7FFF7FFF";
instruction_format <= "10" & "011" & "00010000000100000000";
wait for period;

-- TEST 4 (mixed signs)
rs1 <= x"00000001000000020000000300000004";
rs2 <= x"7FFF80007FFF80007FFF80007FFF8000";
rs3 <= x"80007FFF80007FFF80007FFF80007FFF";
instruction_format <= "10" & "011" & "00000011000000001000";
wait for period;

report "All signed integer multiply - subtract high with saturation tests complete!";
```

Figure 27. Testbench code for SIMSHWS instruction.

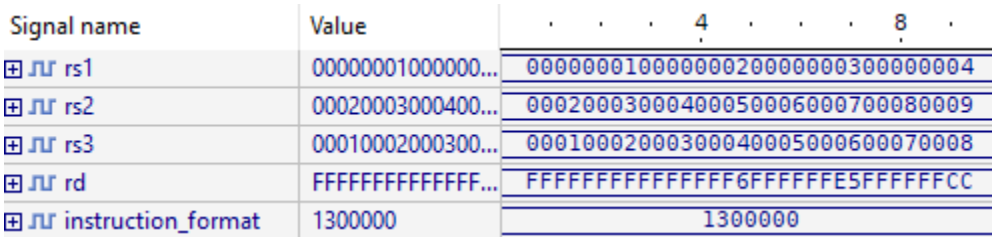


Figure 28. Waveform for test case 1 of SIMSHWS instruction.

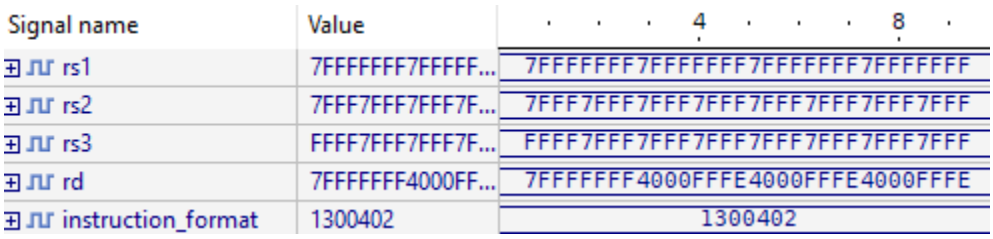


Figure 29. Waveform for test case 2 of SIMSHWS instruction.

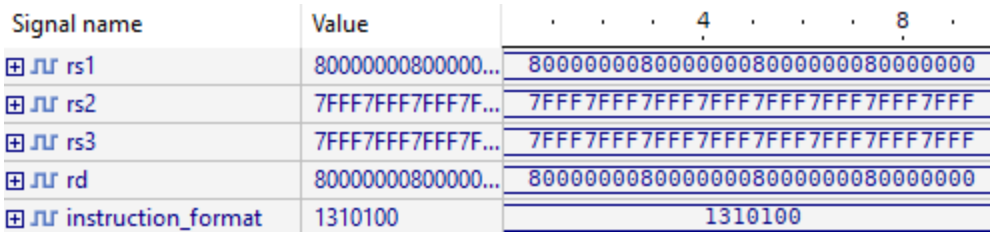


Figure 30. Waveform for test case 3 of SIMSHWS instruction.

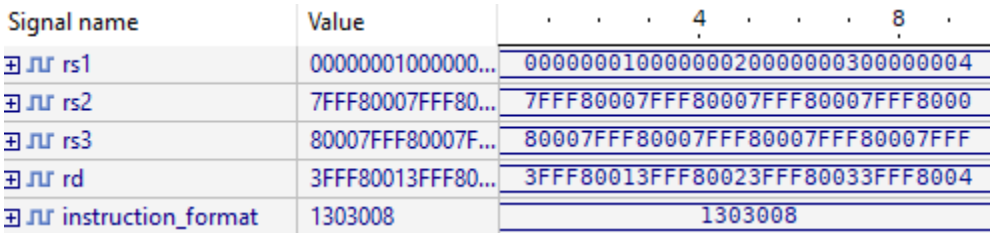


Figure 31. Waveform for test case 4 of SIMSHWS instruction.

Results:

The test cases were designed to test no saturation, positive saturation, negative saturation, and registers with mixed signs. All test cases were successful and produced expected results.

Signed Long Integer Multiply - Add Low with Saturation (SLIMALWS)

```
-- Signed Long Integer Multiply - Add Low with Saturation
when "100" =>
-- Word 0 [63:0]
product_long_64 := signed(rs2(31 downto 0)) * signed(rs3(31 downto 0));
sum_long_64 := resize(signed(rs1(63 downto 0)), 65) + resize(product_long_64, 65);

if sum_long_64 > resize(signed_long_64_max, 65) then
    output(63 downto 0) := STD_LOGIC_VECTOR(signed_long_64_max);
elsif sum_long_64 < resize(signed_long_64_min, 65) then
    output(63 downto 0) := STD_LOGIC_VECTOR(signed_long_64_min);
else
    output(63 downto 0) := STD_LOGIC_VECTOR(sum_long_64(63 downto 0));
end if;

-- Word 1 [127:64]
product_long_64 := signed(rs2(95 downto 64)) * signed(rs3(95 downto 64));
sum_long_64 := resize(signed(rs1(127 downto 64)), 65) + resize(product_long_64, 65);

if sum_long_64 > resize(signed_long_64_max, 65) then
    output(127 downto 64) := STD_LOGIC_VECTOR(signed_long_64_max);
elsif sum_long_64 < resize(signed_long_64_min, 65) then
    output(127 downto 64) := STD_LOGIC_VECTOR(signed_long_64_min);
else
    output(127 downto 64) := STD_LOGIC_VECTOR(sum_long_64(63 downto 0));
end if;
```

Figure 32. Source code for SLIMALWS instruction.

```
-- SLIMALWS --
report "=== STARTING SIGNED LONG INTEGER MULTIPLY - ADD LOW WITH SATURATION TESTBENCH ===";

-- TEST 1 (no saturation)
rs1 <= x"00000000000000010000000000000002";
rs2 <= x"00000002000000030000000400000005";
rs3 <= x"00000003000000040000000500000006";
instruction_format <= "10" & "100" & "00000000000000000000";
wait for period;

-- TEST 2 (+ saturation)
rs1 <= x"7FFFFFFFFFFFFFF7FFFFFFFFFFFFFFF";
rs2 <= x"00000002000000020000000200000002";
rs3 <= x"7FFFFFF7FFFFFFF7FFFFFF7FFFFFFF";
instruction_format <= "10" & "100" & "00000000010000000010";
wait for period;

-- TEST 3 (- saturation)
rs1 <= x"80000000000000008000000000000000";
rs2 <= x"7FFFFFF7FFFFFFF7FFFFFF7FFFFFFF";
rs3 <= x"7FFFFFF7FFFFFFF7FFFFFF7FFFFFFF";
instruction_format <= "10" & "100" & "00010000000100000000";
wait for period;

-- TEST 4 (mixed signs)
rs1 <= x"00000000000000010000000000000002";
rs2 <= x"FFFFFFFF00000001000000000FFFFFFF";
rs3 <= x"0000000200000003FFFFFFFFFFFFFFFE";
instruction_format <= "10" & "100" & "00000011000000001000";
wait for period;

report "All signed long integer multiply - add low with saturation tests complete!";
```

Figure 33. Testbench code for SLIMALWS instruction.

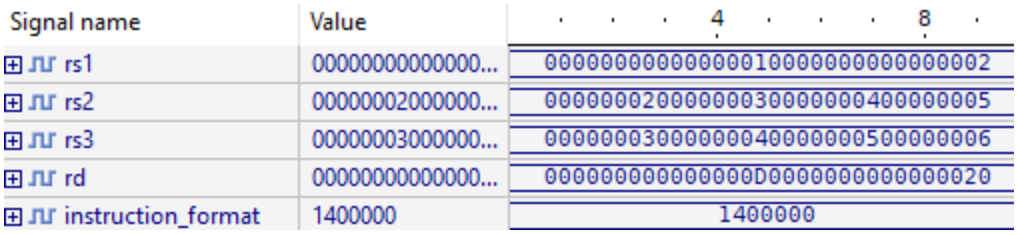


Figure 34. Waveform for test case 1 of SLIMALWS instruction.

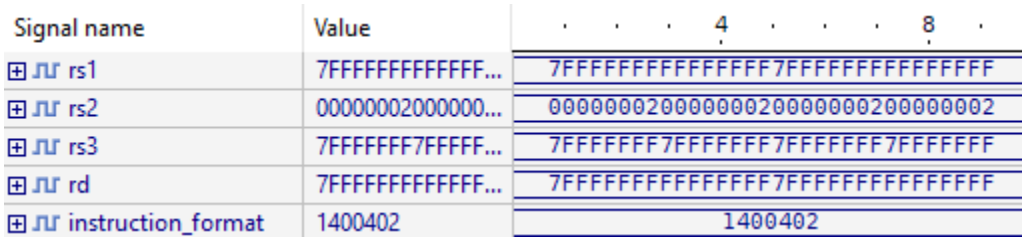


Figure 35. Waveform for test case 2 of SLIMALWS instruction.

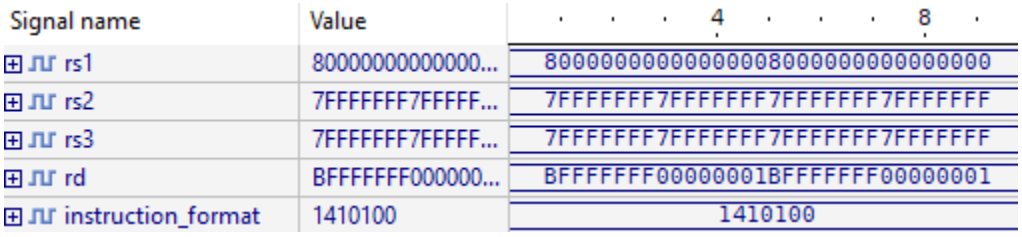


Figure 36. Waveform for test case 3 of SLIMALWS instruction.

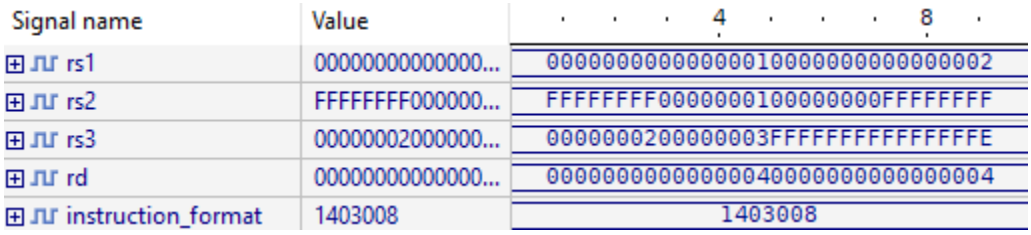


Figure 37. Waveform for test case 4 of SLIMALWS instruction.

Results:

The test cases were designed to test no saturation, positive saturation, negative saturation, and registers with mixed signs. All test cases were successful and produced expected results.

Signed Long Integer Multiply - Add High with Saturation (SLIMAHWS)

```
-- Signed Long Integer Multiply - Add High with Saturation
when "101" =>
-- Word 0 [63:0]
product_long_64 := signed(rs2(63 downto 32)) * signed(rs3(63 downto 32));
sum_long_64 := resize(signed(rs1(63 downto 0)), 65) + resize(product_long_64, 65);
if sum_long_64 > resize(signed_long_64_max, 65) then
    output(63 downto 0) := STD_LOGIC_VECTOR(signed_long_64_max);
elsif sum_long_64 < resize(signed_long_64_min, 65) then
    output(63 downto 0) := STD_LOGIC_VECTOR(signed_long_64_min);
else
    output(63 downto 0) := STD_LOGIC_VECTOR(sum_long_64(63 downto 0));
end if;

-- Word 1 [127:64]
product_long_64 := signed(rs2(127 downto 96)) * signed(rs3(127 downto 96));
sum_long_64 := resize(signed(rs1(127 downto 64)), 65) + resize(product_long_64, 65);
if sum_long_64 > resize(signed_long_64_max, 65) then
    output(127 downto 64) := STD_LOGIC_VECTOR(signed_long_64_max);
elsif sum_long_64 < resize(signed_long_64_min, 65) then
    output(127 downto 64) := STD_LOGIC_VECTOR(signed_long_64_min);
else
    output(127 downto 64) := STD_LOGIC_VECTOR(sum_long_64(63 downto 0));
end if;
```

Figure 38. Source code for SLIMAHWS instruction.

```
-- SLIMAHWS --
report "=== STARTING SIGNED LONG INTEGER MULTIPLY - ADD HIGH WITH SATURATION TESTBENCH ===";

-- TEST 1 (no saturation)
rs1 <= x"0000000000000001000000000000002";
rs2 <= x"0000000200000003000000400000005";
rs3 <= x"0000000300000004000000500000006";
instruction_format <= "10" & "101" & "00000000000000000000";
wait for period;

-- TEST 2 (+ saturation)
rs1 <= x"7FFFFFFFFFFFFFF7FFFFFFFFFFFFFFF";
rs2 <= x"7FFFFFF7FFFFFFF7FFFFFF7FFFFFFF";
rs3 <= x"7FFFFFF7FFFFFFF7FFFFFF7FFFFFFF";
instruction_format <= "10" & "101" & "00000000010000000010";
wait for period;

-- TEST 3 (- saturation)
rs1 <= x"80000000000000008000000000000000";
rs2 <= x"80000000800000008000000080000000";
rs3 <= x"7FFFFFF7FFFFFFF7FFFFFF7FFFFFFF";
instruction_format <= "10" & "101" & "00010000000100000000";
wait for period;

-- TEST 4 (mixed signs)
rs1 <= x"0000000000000001000000000000002";
rs2 <= x"7FFFFFF800000007FFFFFF80000000";
rs3 <= x"800000007FFFFFF800000007FFFFFF";
instruction_format <= "10" & "101" & "00000011000000001000";
wait for period;

report "All signed long integer multiply - add high with saturation tests complete!";
```

Figure 39. Testbench code for SLIMAHWS instruction.

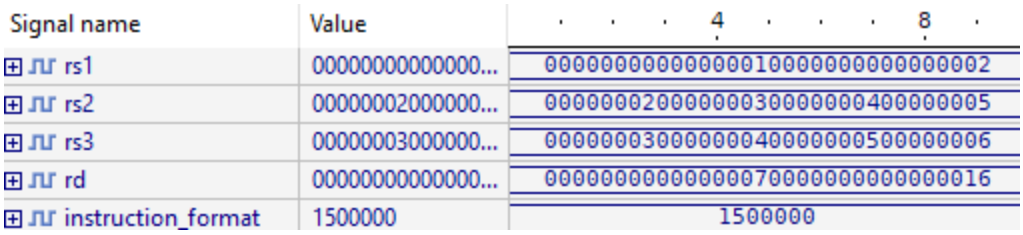


Figure 40. Waveform for test case 1 of SLIMAHWS instruction.

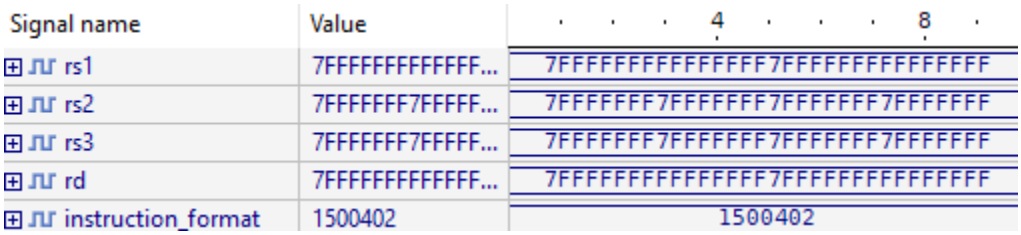


Figure 41. Waveform for test case 2 of SLIMAHWS instruction.

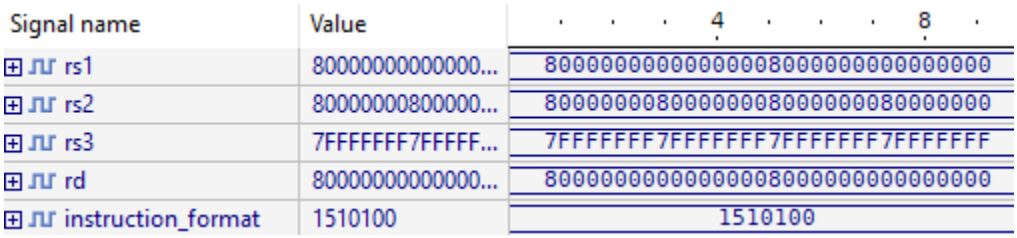


Figure 42. Waveform for test case 3 of SLIMAHWS instruction.

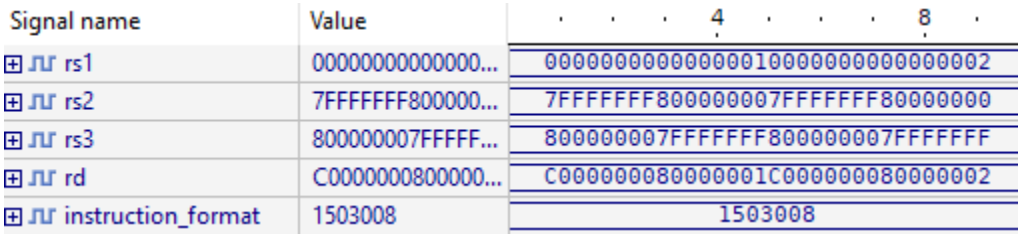


Figure 43. Waveform for test case 4 of SLIMAHWS instruction.

Results:

The test cases were designed to test no saturation, positive saturation, negative saturation, and registers with mixed signs. All test cases were successful and produced expected results.

Signed Long Integer Multiply - Subtract Low with Saturation (SLIMSLWS)

```
-- Signed Long Integer Multiply - Subtract Low with Saturation
when "110" =>
-- Word 0 [63:0]
product_long_64 := signed(rs2(31 downto 0)) * signed(rs3(31 downto 0));
diff_long_64 := resize(signed(rs1(63 downto 0)), 65) - resize(product_long_64, 65);
if diff_long_64 > resize(signed_long_64_max, 65) then
    output(63 downto 0) := STD_LOGIC_VECTOR(signed_long_64_max);
elsif diff_long_64 < resize(signed_long_64_min, 65) then
    output(63 downto 0) := STD_LOGIC_VECTOR(signed_long_64_min);
else
    output(63 downto 0) := STD_LOGIC_VECTOR(diff_long_64(63 downto 0));
end if;

-- Word 1 [127:64]
product_long_64 := signed(rs2(95 downto 64)) * signed(rs3(95 downto 64));
diff_long_64 := resize(signed(rs1(127 downto 64)), 65) - resize(product_long_64, 65);
if diff_long_64 > resize(signed_long_64_max, 65) then
    output(127 downto 64) := STD_LOGIC_VECTOR(signed_long_64_max);
elsif diff_long_64 < resize(signed_long_64_min, 65) then
    output(127 downto 64) := STD_LOGIC_VECTOR(signed_long_64_min);
else
    output(127 downto 64) := STD_LOGIC_VECTOR(diff_long_64(63 downto 0));
end if;
```

Figure 44. Source code for SLIMSLWS instruction.

```
--SLIMSLWS --
report "=== STARTING SIGNED LONG INTEGER MULTIPLY - SUBTRACT LOW WITH SATURATION TESTBENCH ===";

-- TEST 1 (no saturation)
rs1 <= x"00000000000000002000000000000004";
rs2 <= x"00000002000000030000000400000005";
rs3 <= x"00000003000000040000000500000006";
instruction_format <= "10" & "110" & "00000000000000000000";
wait for period;

-- TEST 2 (+ saturation)
rs1 <= x"7FFFFFFFFFFFFFFF7FFFFFFFFFFFFFFF";
rs2 <= x"00000000FFFFFFFF00000000FFFFFFFF";
rs3 <= x"000000007FFFFFFF000000007FFFFFFF";
instruction_format <= "10" & "110" & "00000000000000000000";
wait for period;

-- TEST 3 (- saturation)
rs1 <= x"80000000000000008000000000000000";
rs2 <= x"7FFFFFFF7FFFFFFF7FFFFFFF7FFFFFFF";
rs3 <= x"7FFFFFFF7FFFFFFF7FFFFFFF7FFFFFFF";
instruction_format <= "10" & "110" & "00000000000000000000";
wait for period;

-- TEST 4 (mixed signs)
rs1 <= x"00000000000000010000000000000002";
rs2 <= x"FFFFFFFF0000000100000000FFFFFFFF";
rs3 <= x"0000000200000003FFFFFFFFFFFFFFFE";
instruction_format <= "10" & "110" & "00000000000000000000";
wait for period;

report "All signed long integer multiply - subtract low with saturation tests complete!";
```

Figure 45. Testbench code for SLIMSLWS instruction.

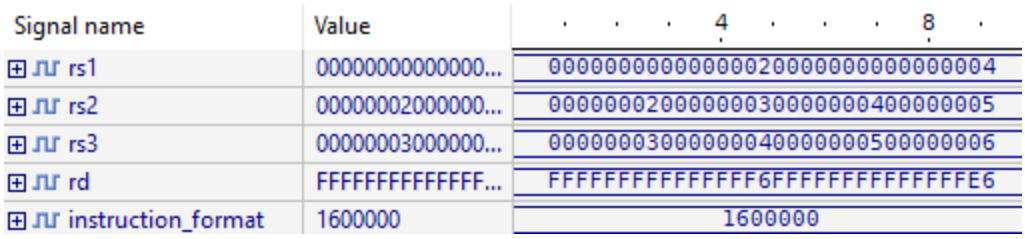


Figure 46. Waveform for test case 1 of SLIMSLWS instruction.

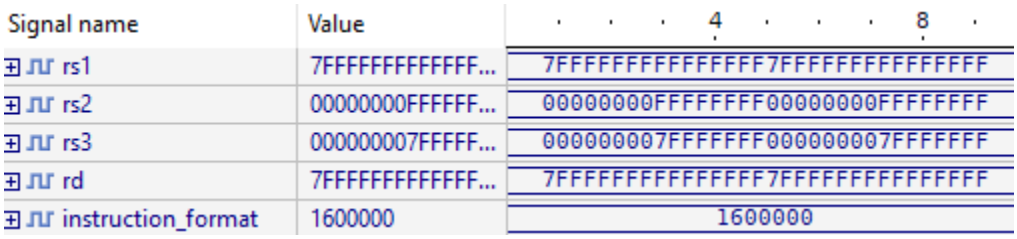


Figure 47. Waveform for test case 2 of SLIMSLWS instruction.

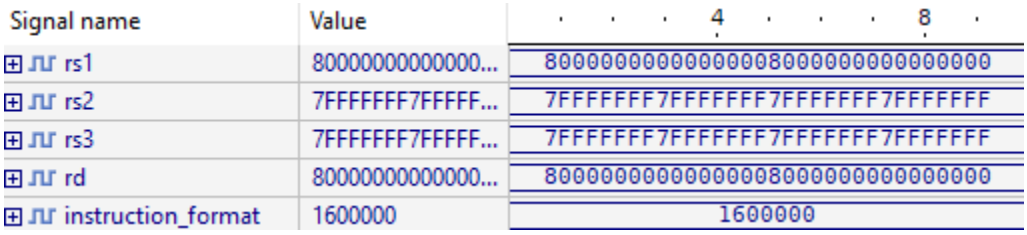


Figure 48. Waveform for test case 3 of SLIMSLWS instruction.

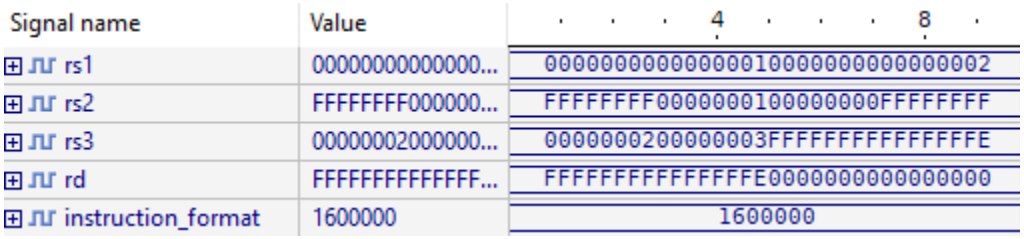


Figure 49. Waveform for test case 4 of SLIMSLWS instruction.

Results:

The test cases were designed to test no saturation, positive saturation, negative saturation, and registers with mixed signs. All test cases were successful and produced expected results.

Signed Long Integer Multiply - Subtract High with Saturation (SLIMSHWS)

```
-- Signed Long Integer Multiply - Subtract High with Saturation
when "111" =>
-- Word 0 [63:0]
product_long_64 := signed(rs2(63 downto 32)) * signed(rs3(63 downto 32));
diff_long_64 := resize(signed(rs1(63 downto 0)), 65) - resize(product_long_64, 65);
if diff_long_64 > resize(signed_long_64_max, 65) then
    output(63 downto 0) := STD_LOGIC_VECTOR(signed_long_64_max);
elsif diff_long_64 < resize(signed_long_64_min, 65) then
    output(63 downto 0) := STD_LOGIC_VECTOR(signed_long_64_min);
else
    output(63 downto 0) := STD_LOGIC_VECTOR(diff_long_64(63 downto 0));
end if;

-- Word 1 [127:64]
product_long_64 := signed(rs2(127 downto 96)) * signed(rs3(127 downto 96));
diff_long_64 := resize(signed(rs1(127 downto 64)), 65) - resize(product_long_64, 65);
if diff_long_64 > resize(signed_long_64_max, 65) then
    output(127 downto 64) := STD_LOGIC_VECTOR(signed_long_64_max);
elsif diff_long_64 < resize(signed_long_64_min, 65) then
    output(127 downto 64) := STD_LOGIC_VECTOR(signed_long_64_min);
else
    output(127 downto 64) := STD_LOGIC_VECTOR(diff_long_64(63 downto 0));
end if;
```

Figure 50. Source code for SLIMSHWS instruction.

```
--SLIMSHWS --
report "=== STARTING SIGNED LONG INTEGER MULTIPLY - SUBTRACT HIGH WITH SATURATION TESTBENCH ===";

-- TEST 1 (no saturation)
rs1 <= x"00000000000000020000000000000004";
rs2 <= x"00000002000000030000000400000005";
rs3 <= x"00000003000000040000000500000006";
instruction_format <= "10" & "111" & "00000000000000000000";
wait for period;

-- TEST 2 (+ saturation)
rs1 <= x"7FFFFFFFFFFFFFF7FFFFFFFFFFFFFFF";
rs2 <= x"00000000FFFFFFF00000000FFFFFFF";
rs3 <= x"000000007FFFFFF000000007FFFFFF";
instruction_format <= "10" & "111" & "00000000000000000000";
wait for period;

-- TEST 3 (- saturation)
rs1 <= x"80000000000000008000000000000000";
rs2 <= x"7FFFFFF7FFFFFFF7FFFFFF7FFFFFFF";
rs3 <= x"7FFFFFF7FFFFFFF7FFFFFF7FFFFFFF";
instruction_format <= "10" & "111" & "00000000000000000000";
wait for period;

-- TEST 4 (mixed signs)
rs1 <= x"00000000000000010000000000000002";
rs2 <= x"7FFFFFFF800000007FFFFFFF80000000";
rs3 <= x"800000007FFFFFFF800000007FFFFFFF";
instruction_format <= "10" & "111" & "00000000000000000000";
wait for period;

report "All signed long integer multiply - subtract high with saturation tests complete!";
```

Figure 51. Source code for SLIMSHWS instruction.

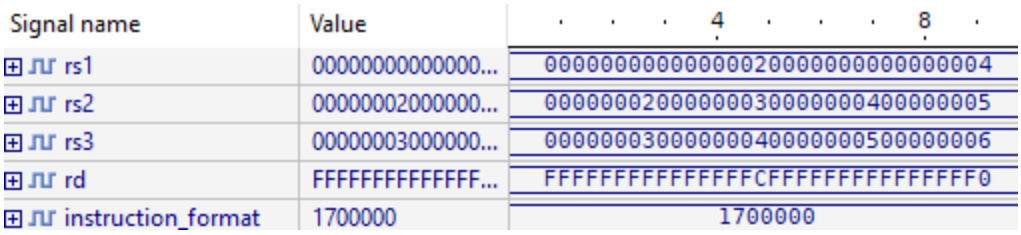


Figure 52. Waveform for test case 1 of SLIMSHWS instruction.

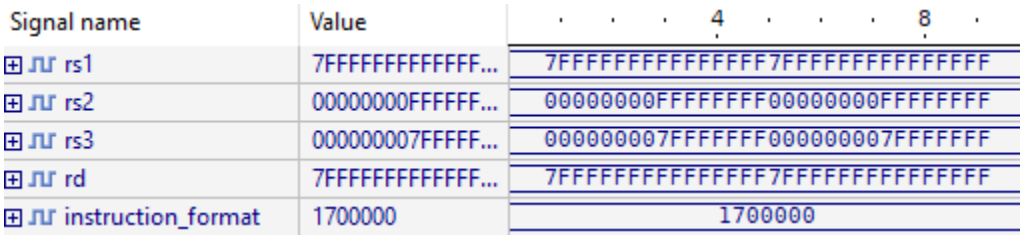


Figure 53. Waveform for test case 2 of SLIMSHWS instruction.

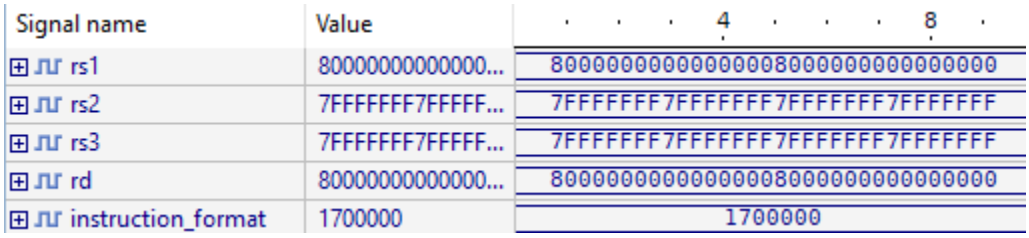


Figure 54. Waveform for test case 3 of SLIMSHWS instruction.

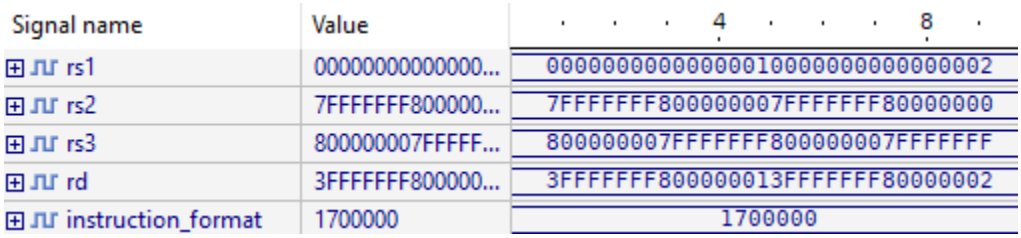


Figure 55. Waveform for test case 4 of SLIMSHWS instruction.

Results:

The test cases were designed to test no saturation, positive saturation, negative saturation, and registers with mixed signs. All test cases were successful and produced expected results.

R3 Instruction Format

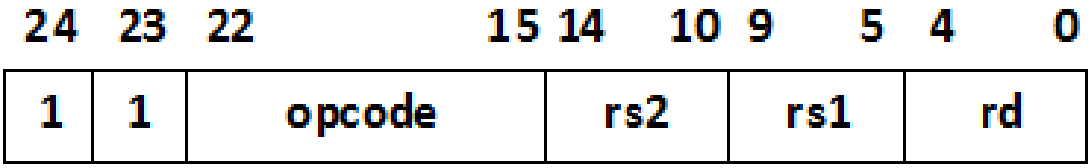


Figure 56. The instruction format for R3 instructions.

Bit Range	Field Name	Description
24	Opcode (1)	First bit to identify the instruction format.
23	Opcode (1)	The second bit is used to verify this is the R3 instruction format ('11').
22-15	opcode	Selects the specific operation to be executed on the data from source registers rs1 and rs2.
14-10	rs2	Second source register.
9-5	rs1	First source register.
4-0	rd	Destination register.

Table 6. Table explaining the bits of the R3 instruction format.

Opcode [22:15]	Instruction	Description
xxxx0000	NOP: no operation	Performs no operation. The instruction must not modify any register or memory contents.
xxxx0001	SHRHI: shift right halfword immediate	Performs a logical right shift on each 16-bit halfword within the 128-bit register rs1, by the amount specified in the 4 least-significant bits of the instruction field of rs2 [13:10]. Each shifted result is placed in the corresponding

		halfword of rd.
xxxx0010	AU: add word unsigned	Performs unsigned 32-bit addition for each word field of rs1 and rs2, producing four independent 32-bit results stored in rd.
xxxx0011	CNT1H: count 1s in halfword	Counts the number of ‘1’ bits in each 16-bit halfword of rs1, and writes the count into the corresponding halfword of rd.
xxxx0100	AHS: add halfword saturated	Performs signed 16-bit addition with saturation between corresponding halfwords of rs1 and rs2. Overflow or underflow results are clamped to the maximum or minimum 16-bit signed value.
xxxx0101	OR: bitwise logical or	Computes the bitwise OR of corresponding bits in rs1 and rs2, storing the result in rd.
xxxx0110	BCW: broadcast word	Replicates the leftmost 32-bit word of rs1 into all four 32-bit word fields of rd, effectively broadcasting a single word across the entire 128-bit register.
xxxx0111	MAXWS: max signed word	Compares each pair of 32-bit signed words from rs1 and rs2, placing the maximum value of each pair into the corresponding field of rd.
xxxx1000	MINWS: min signed word	Compares each pair of 32-bit signed words from rs1 and rs2, placing the minimum value of each pair into the corresponding field of rd.
xxxx1001	MLHU: multiply low	Multiplies the lower 16 bits

	unsigned	of each 32-bit word in rs1 and rs2 as unsigned values, placing each 32-bit product in the corresponding field of rd.
xxxx1010	MLHCU: multiply low by constant unsigned	Multiplies the lower 16 bits of each 32-bit word in rs1 by a 5-bit immediate constant specified in the rs2 field [14:10]. All operands are treated as unsigned, and the 32-bit products are stored in the corresponding 32-bit fields of rd.
xxxx1011	AND: bitwise logical and	Computes the bitwise AND of corresponding bits in rs1 and rs2, storing the result in rd.
xxxx1100	CLZW: count leading zeroes in words	Counts the number of leading zeros in each 32-bit word of rs1. If a word is entirely zero, the result is 32. Each count is placed into the corresponding 32-bit word of rd.
xxxx1101	ROTW: rotate bits in word	Rotates each 32-bit word in rs1 right by the value in the 5 least-significant bits of the corresponding word in rs2, with bits wrapped around to form a circular rotation.
xxxx1110	SFWU: subtract from word unsigned	Performs unsigned 32-bit subtraction of rs1 from rs2, storing the result (rd = rs2 – rs1) in the corresponding 32-bit fields of rd.
xxxx1111	SFHS: subtract from halfword saturated	Performs signed 16-bit subtraction with saturation, subtracting each halfword of rs1 from rs2 (rd = rs2 – rs1).

Table 7. Table explaining all instructions in R3 instruction format.

NOP: no operation

```
-- NOP
when "0000" =>
rd <= rd;
```

Figure 57. Source code for NOP instruction.

```
-- NOP --
report "=== STARTING NOP TESTBENCH ===";

-- TEST 1
-- Using SIMSHWS to output a rd and seeing if rd stays the same in nop
rs1 <= x"00000001000000020000000300000004";
rs2 <= x"7FFF80007FFF80007FFF80007FFF8000";
rs3 <= x"80007FFF80007FFF80007FFF80007FFF";
instruction_format <= "10" & "011" & "0000011000000001000";
wait for period;

rs1 <= x"11112222333344445555666677778888";
rs2 <= x"9999AAAAFFFFEEEE7777555544443333";
rs3 <= x"00000000000000000000000000000000"; -- no longer needed for R3 instructions
instruction_format <= "11" & "00000000" & "000000000000000";
wait for period;

report "All nop tests complete!";
```

Figure 58. Testbench code for NOP instruction.

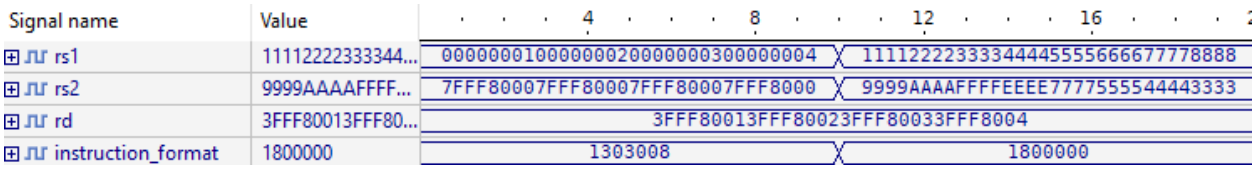


Figure 59. Waveform for test case 1 of NOP instruction.

Results:

We used the SIMSHWS instruction in the beginning to begin the test case with a rd value that is not uninitialized. Thus, from 0-10ns, that is the result of the SIMSHWS instruction. 10-20ns displays no operation and thus, the same rd value extends to 20ns. The test case was successful and produced expected results.

SHRHI: shift right halfword immediate

```
-- SHRHI
when "0001" =>
shift := to_integer(unsigned(instruction_format(13 downto 10)));
    for i in 0 to 7 loop
        output((i*16+15) downto (i*16)) := std_logic_vector(shift_right(unsigned(rs1((i*16+15) downto (i*16))), shift));
    end loop;
rd <= output;
```

Figure 60. Source code for SHRHI instruction.

```
-- SHRHI --
report "=== STARTING SHRHI TESTBENCH ===";

-- TEST 1 (shift 8)
rs1 <= x"ABCD1234FFFF00FF000080007FFF7FFF";
rs2 <= x"00000000000000000000000000000000"; -- value not important here so reset
instruction_format <= "11" & "00010001" & "010000000000000";
wait for period;

-- TEST 2 (max shift)
rs1 <= x"FFFF0001800000017FFF800000010002";
instruction_format <= "11" & "00010001" & "111110000000000";
wait for period;

-- TEST 3 (mixed +/-)
rs1 <= x"80007FFF00008000FFFF000180007FFF";
instruction_format <= "11" & "00010001" & "000110000000000";
wait for period;

report "All shrhi tests complete!";
```

Figure 61. Testbench code for SHRHI instruction.

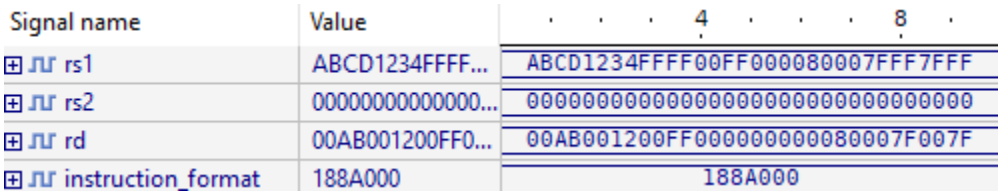


Figure 62. Waveform for test case 1 of SHRHI instruction.

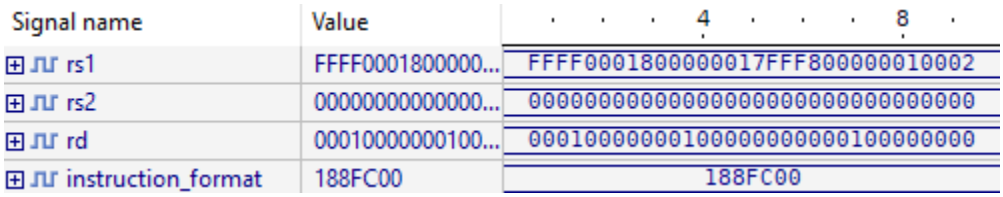


Figure 63. Waveform for test case 2 of SHRHI instruction.

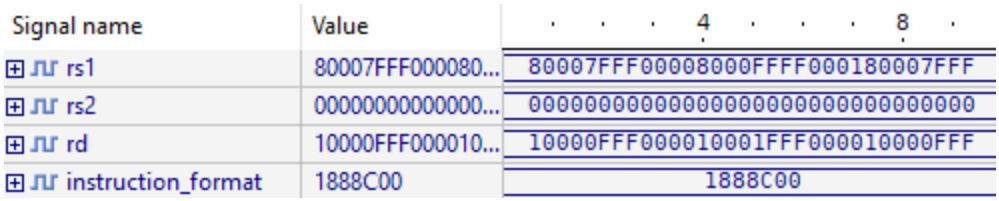


Figure 64. Waveform for test case 3 of SHRHI instruction.

Results:

rs2 is cleared because the value in this register is irrelevant for this operation. Only the least four significant bits of the instruction field of rs2 matter here, NOT the least four significant bits of the 128-bit value of rs2. We tested cases shifting by eight bits, a max shift, and registers with mixed signs. The test cases were successful and produced expected results.

AU: add word unsigned

```
--AU
when "0010" =>
for i in 0 to 3 loop
    w1 := unsigned(rs1((i*32+31) downto (i*32)));
    w2 := unsigned(rs2((i*32+31) downto (i*32)));
    result := w1 + w2;
    reg_result((i*32+31) downto (i*32)) := signed(result);
end loop;

rd <= std_logic_vector(reg_result);
```

Figure 65. Source code for AU instruction.

```
-- AU --
report "=== STARTING AU TESTBENCH ===";

-- TEST 1 (no overflow)
rs1 <= x"FFFF0001FFF10010ABCD98765432FEDC";
rs2 <= x"00000000000000000000000000000004";
instruction_format <= "11" & "01100010" & "0000000000000000";
wait for period;

-- TEST 2 (overflow)
rs1 <= x"FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF";
rs2 <= x"00000001000000010000000100000001";
instruction_format <= "11" & "00000010" & "0000000000000000";
wait for period;

report "All au tests complete!";
```

Figure 66. Testbench code for AU instruction.

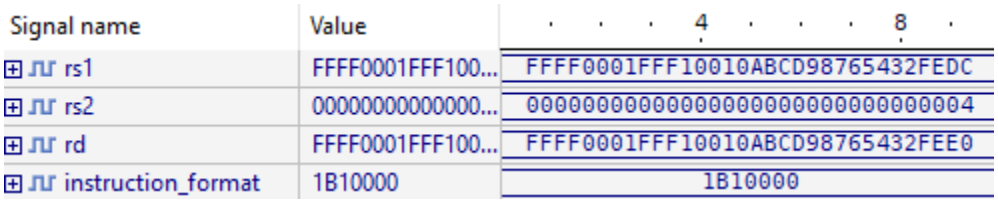


Figure 67. Waveform for test case 1 of AU instruction.

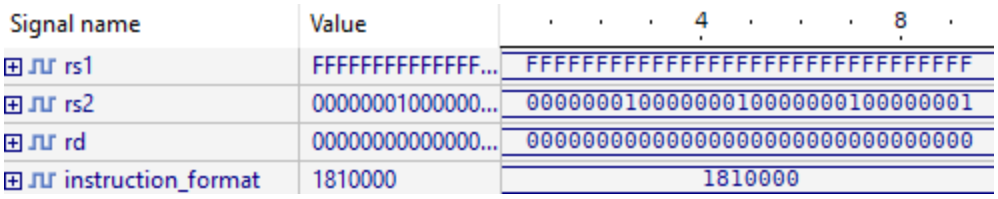


Figure 68. Waveform for test case 2 of AU instruction.

Results:

We tested cases with no overflow and overflow. The test cases were successful and produced expected results.

CNT1H: count 1s in halfword

```
--CNT1H
when "0011" =>
cnt := 0;

for i in 0 to 7 loop
    cnt := 0;
    for bit_i in 0 to 15 loop
        if rs1(i*16 + bit_i) = '1' then
            cnt := cnt + 1;
        end if;
    end loop;
    reg_result((i*16+15) downto (i*16)) := to_signed(cnt, 16);
end loop;
rd <= std_logic_vector(reg_result);
```

Figure 69. Source code for CNT1H instruction.

```
-- CNT1H --
report "=== STARTING CNT1H TESTBENCH ===";

-- TEST 1 (all ones)
rs1 <= x"FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF";
rs2 <= x"00000000000000000000000000000000"; -- not needed so reset
instruction_format <= "11" & "00000011" & "0000000000000000";
wait for period;

-- TEST 2 (alternating)
rs1 <= x"AAAA5555AAAA5555AAAA5555AAAA5555";
instruction_format <= "11" & "00000011" & "0000000000000000";
wait for period;

report "All cnt1h tests complete!";
```

Figure 70. Testbench code for CNT1H instruction.

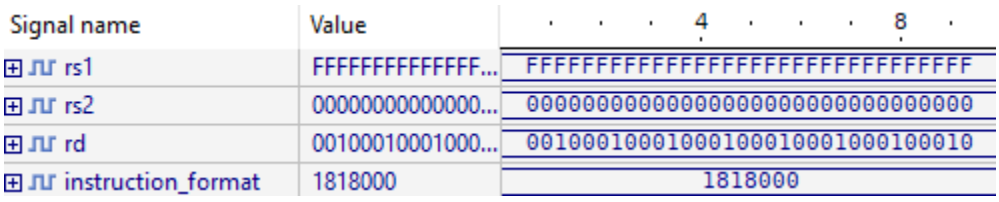


Figure 71. Waveform for test case 1 of CNT1H instruction.

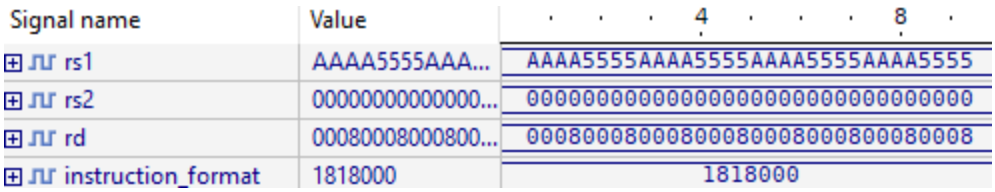


Figure 72. Waveform for test case 2 of CNT1H instruction.

Results:

rs2 is cleared because the value in this register is irrelevant for this operation. We tested cases with all bits one and alternating bits. The test cases were successful and produced expected results.

AHS: add halfword saturated

```
--AHS
when "0100" =>
-- Extract 16 bits from each reg, starting from 15 to 0
     for i in 0 to 7 loop
          reg1 := signed(rs1((i*16 + 15) downto (i*16)));
          reg2 := signed(rs2((i*16 + 15) downto (i*16)));

     -- add and saturate
     sum := resize(reg1, 17) + resize(reg2, 17);
     if sum > to_signed(32767, 17) then
          -- edit the respective bit positions  to_signed(integer_value, length)
          reg_result((i*16 + 15) downto (i*16)) :=  to_signed(32767, 16);

     elsif sum < to_signed(-32768, 17) then
          reg_result((i*16 + 15) downto (i*16)) := to_signed(-32768,16);

     else -- we can use the respective sums
          reg_result((i*16 + 15) downto (i*16)) := resize(sum,16);
     end if;
end loop;
rd <= std_logic_vector(reg_result);
```

Figure 73. The instruction format for AHS instruction

```
-- AHS --
report "=== STARTING AHS TESTBENCH ===";

-- TEST 1 (no saturation)
rs1 <= x"0001000200030004000500060070008";
rs2 <= x"0008000700060005000400030020001";
instruction_format <= "11" & "00000100" & "000010000010000";
wait for period;

 TEST 2 (+ overflow)
rs1 <= x"7FFF7FFF7FFF7FFF7FFF7FFF7FFF7FFF";
rs2 <= x"0001000100010001000100010001001";
instruction_format <= "11" & "00000100" & "000000000000000";
wait for period;

-- TEST 3 (- overflow)
rs1 <= x"8000800080008000800080008008000";
rs2 <= x"FFFF8000FFFF8000FFFF8000FFFF8000";
instruction_format <= "11" & "00000100" & "000000000000000";
wait for period;

report "All ahs tests complete!";
```

Figure 74. Testbench code for AHS instruction.

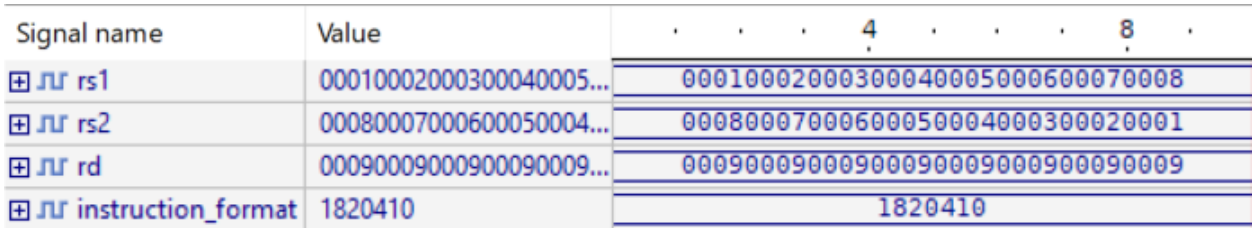


Figure 75. Waveform for test case 1 of AHS instruction.

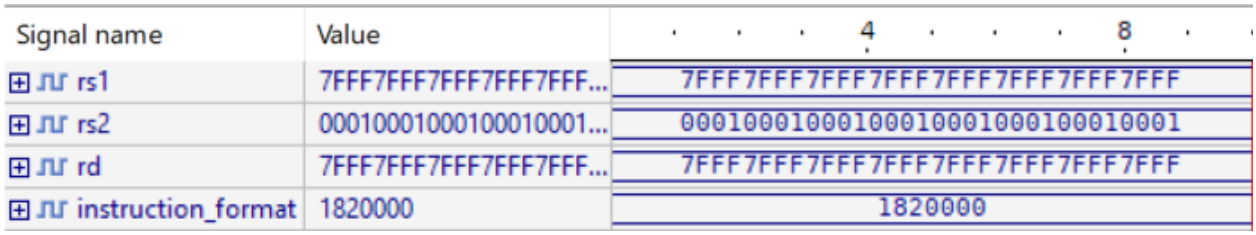


Figure 76. Waveform for test case 2 of AHS instruction.

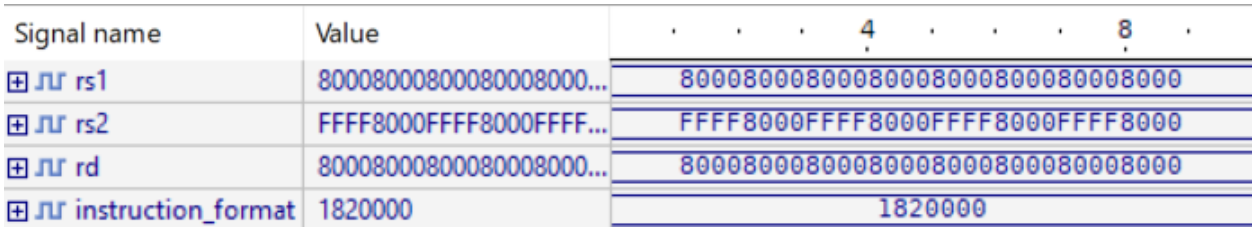


Figure 77. Waveform for test case 3 of AHS instruction.

Results:

The test cases were designed to test no saturation, positive saturation, and negative saturation. All test cases were successful and produced expected results.

OR: bitwise logical or

```
--OR
when "0101" =>
rd <= rs1 or rs2;
```

Figure 78. Source code for OR instruction.

```
-- OR --
report "=== STARTING OR TESTBENCH ===";

-- TEST 1 (random)
rs1 <= x"12345678ABCDEF00FF00FF00FF00FF00";
rs2 <= x"00FF00FF00FF00FF00FF00FF00FF00FF";
instruction_format <= "11" & "00000101" & "000001000000000";
wait for period;

-- TEST 2 (alternating bits)
rs1 <= x"F0F0F0F0F0F0F0F0F0F0F0F0F0F0F0F0";
rs2 <= x"0F0F0F0F0F0F0F0F0F0F0F0F0F0F0F0F";
instruction_format <= "11" & "00000101" & "000000000000001";
wait for period;

report "All or tests complete!";
```

Figure 79. Testbench code for OR instruction.

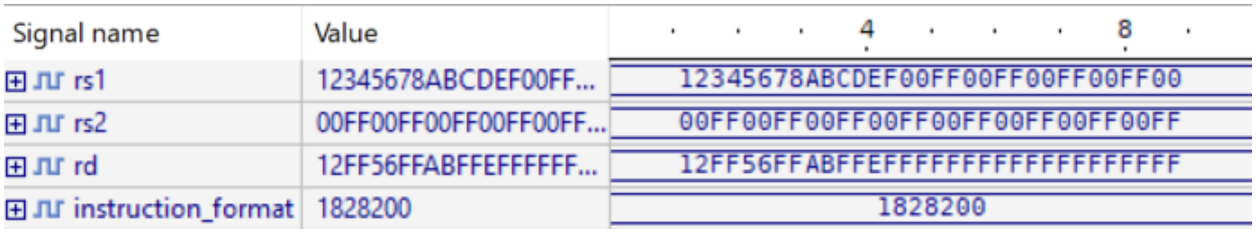


Figure 80. Waveform for test case 1 of OR instruction.

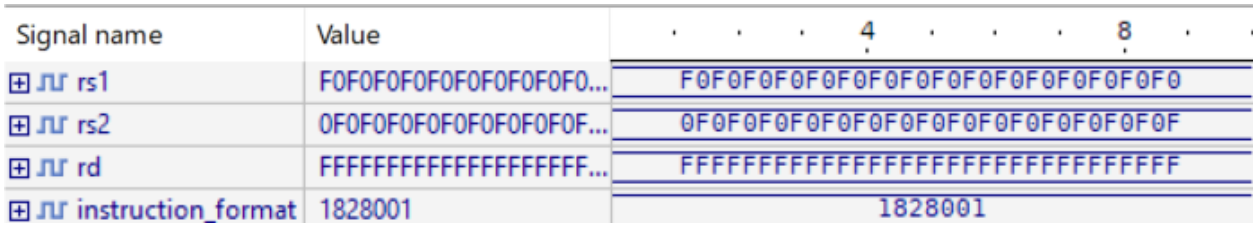


Figure 81. Waveform for test case 2 of OR instruction.

Results:

The test cases were designed to test random and alternating bits. All test cases were successful and produced expected results.

BCW: broadcast word

```
-- BCW --
report "=== STARTING BCW TESTBENCH ===";

-- TEST 1 (random)
rs1 <= x"DEADBEEFCAFEBABE0123456789ABCDEF";
rs2 <= x"00000000000000000000000000000000"; -- not needed so reset
instruction_format <= "11" & "00000110" & "000000000000001";
wait for period;

-- TEST 2 (alternating bits)
rs1 <= x"A5A5A5A55A5A5A5AFFFFFFFF00000000";
instruction_format <= "11" & "00000110" & "000000000000001";
wait for period;

report "All bcw tests complete!";
```

Figure 82. Source code for BCW instruction.

```
--BCW
when "0110" =>
rd(31 downto 0) <= rs1(127 downto 96);
rd(63 downto 32) <= rs1(127 downto 96);
rd(95 downto 64) <= rs1(127 downto 96);
rd(127 downto 96) <= rs1(127 downto 96);
```

Figure 83. Testbench code for BCW instruction.

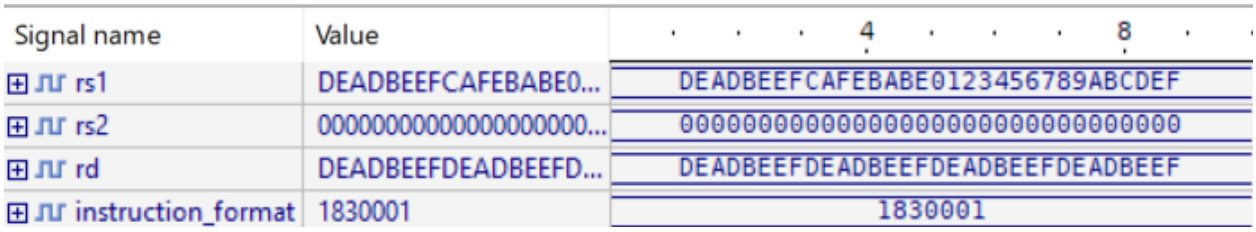


Figure 84. Waveform for test case 1 of BCW instruction.

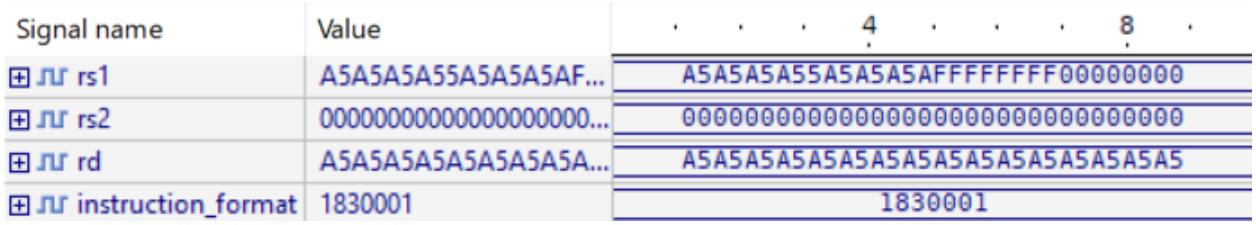


Figure 85. Waveform for test case 2 of BCW instruction.

Results:

rs2 was reset because the value is irrelevant in this instruction. The test cases were designed to test random and alternating bits. All test cases were successful and produced expected results.

MAXWS: max signed word

```
--MAXWS
when "0111" =>
     for i in 0 to 3 loop
          reg3 := signed(rs1((i*32+31) downto (i*32))); -- extract the 32 bits word from rs1
          reg4 := signed(rs2((i*32+31) downto (i*32))); -- extract the 32 bits word from rs2

     if reg3 > reg4 then      -- if the 32 bits is rs1 is greater, store rs1
          reg_result((i*32+31) downto (i*32)) := reg3;
     else                     -- else rs2 is greater, store rs2
          reg_result((i*32+31) downto (i*32)) := reg4;
     end if;
end loop;
rd <= std_logic_vector(reg_result);
```

Figure 86. Source code for MAXWS instruction.

```
-- MAXWS --
report "=== STARTING MAXWS TESTBENCH ===";

-- TEST 1 (+)
rs1 <= x"00000001000000020000000300000004";
rs2 <= x"00000004000000030000000200000001";
instruction_format <= "11" & "00000111" & "000000000000000";
wait for period;

-- TEST 2 (mixed signs)
rs1 <= x"80000000FFFFFFF7FFFFFF00000000";
rs2 <= x"7FFFFFFF8000000000000000FFFFFFFF";
instruction_format <= "11" & "00000111" & "000000000000000";
wait for period;

-- TEST 3 (largest and smallest signed)
rs1 <= x"7FFFFFFF800000007FFFFFF80000000";
rs2 <= x"800000007FFFFFF800000007FFFFFF";
instruction_format <= "11" & "00000111" & "000000000000000";
wait for period;

report "All maxws tests complete!";
```

Figure 87. Testbench code for MAXWS instruction.

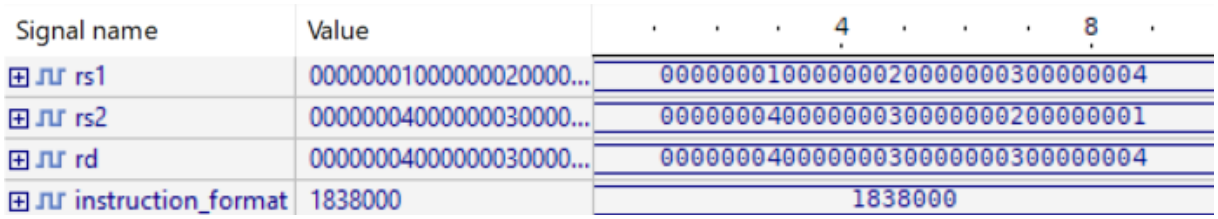


Figure 88. Waveform for test case 1 of MAXWS instruction.

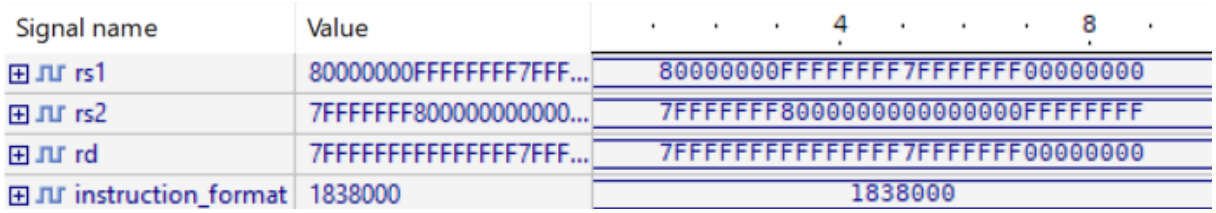


Figure 89. Waveform for test case 2 of MAXWS instruction.

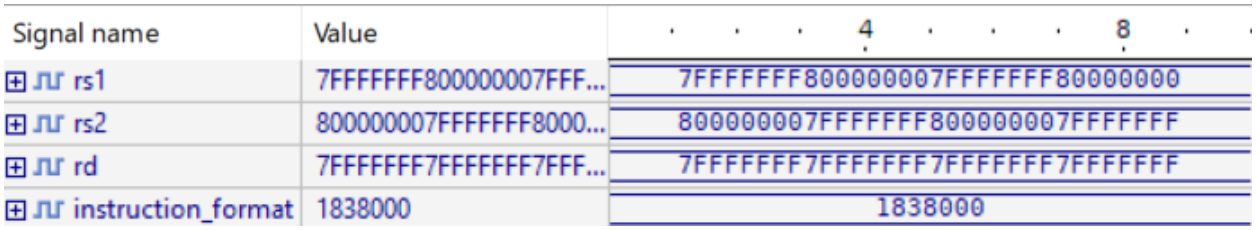


Figure 90. Waveform for test case 3 of MAXWS instruction.

Results:

The test cases were designed to test registers with positive, mixed sign, and largest/smallest signed values. All test cases were successful and produced expected results.

MINWS: min signed word

```
--MINWS
when "1000" =>
for i in 0 to 3 loop
         reg3 := signed(rs1((i*32+31) downto (i*32))); -- extract the 32 bits word from rs1
         reg4 := signed(rs2((i*32+31) downto (i*32))); -- extract the 32 bits word from rs2

     if reg3 < reg4 then      -- if the 32 bits is rs1 is greater, store rs1
          reg_result((i*32+31) downto (i*32)) := reg3;
     else                -- else rs2 is greater, store rs2
          reg_result((i*32+31) downto (i*32)) := reg4;
     end if;
end loop;
rd <= std_logic_vector(reg_result);
```

Figure 91. Source code for MINWS instruction.

```
-- MINWS --
report "=== STARTING MINWS TESTBENCH ===";

-- TEST 1 (+)
rs1 <= x"00000001000000020000000300000004";
rs2 <= x"00000004000000030000000200000001";
instruction_format <= "11" & "00001000" & "000000000000000";
wait for period;

-- TEST 2 (mixed signs)
rs1 <= x"80000000FFFFFFF7FFFFFF00000000";
rs2 <= x"7FFFFFFF80000000000000000FFFFFFF";
instruction_format <= "11" & "00001000" & "000000000000000";
wait for period;

-- TEST 3 (largest and smallest signed)
rs1 <= x"7FFFFFFF800000007FFFFFF80000000";
rs2 <= x"800000007FFFFFF800000007FFFFFFF";
instruction_format <= "11" & "00001000" & "000000000000000";
wait for period;

report "All minws tests complete!";
```

Figure 92. Testbench code for MINWS instruction.

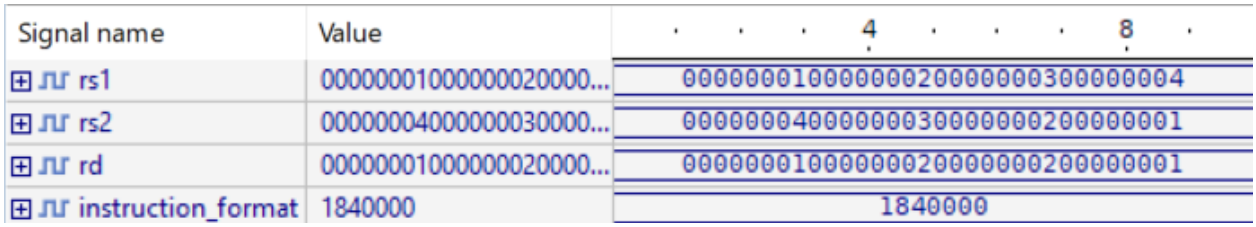


Figure 93. Waveform for test case 1 of MINWS instruction.

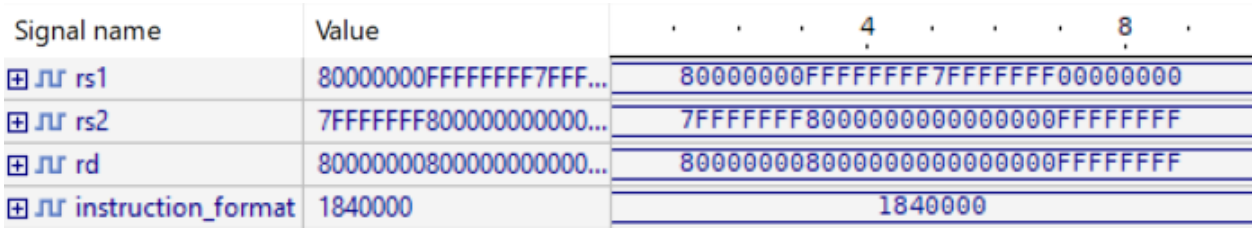


Figure 94. Waveform for test case 2 of MINWS instruction.

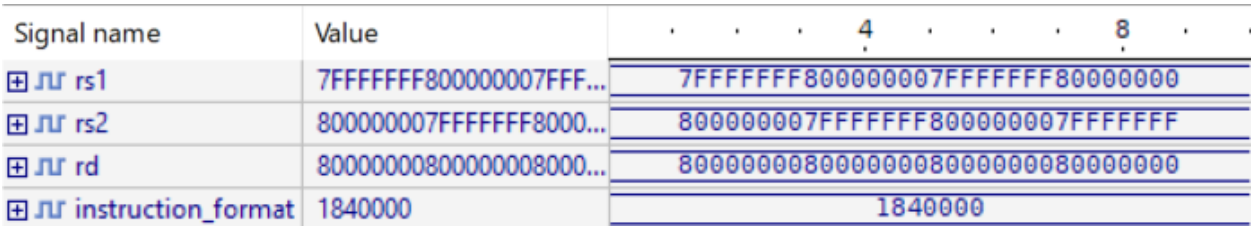


Figure 95. Waveform for test case 3 of MINWS instruction.

Results:

The test cases were designed to test registers with positive, mixed sign, and largest/smallest signed values. All test cases were successful and produced expected results.

MLHU: multiply low unsigned

```
--MLHU
when "1001" =>
for i in 0 to 3 loop

    --convert 16 rightmost bits into val
    half_rs1 := unsigned(rs1((i*32+15) downto (i*32)));
    half_rs2 := unsigned(rs2((i*32+15) downto (i*32)));

    -- Multiply the 16 rightmost
    product := resize((half_rs1 * half_rs2), 32);
    rd((i*32 + 31) downto (i*32)) <= std_logic_vector(product);
end loop;
```

Figure 96. Source code for MLHU instruction.

```
-- MLHU --
report "=== STARTING MLHU TESTBENCH ===";

-- TEST 1 (no overflow)
rs1 <= x"0001000200030004000100020003004";
rs2 <= x"0005000600070008000500060007008";
instruction_format <= "11" & "00001001" & "000000000000000";
wait for period;

-- TEST 2 (mixed bits)
rs1 <= x"00010000FFFF123400010000FFFF1234";
rs2 <= x"FFFF0001ABCD1111FFFF0001ABCD1111";
instruction_format <= "11" & "11111001" & "000000000000000";
wait for period;

-- TEST 3 (max unsigned, overflow)
rs1 <= x"FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF";
rs2 <= x"FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF";
instruction_format <= "11" & "00001001" & "000000000000000";
wait for period;

report "All mlhu tests complete!";
```

Figure 97. Testbench code for MLHU instruction.

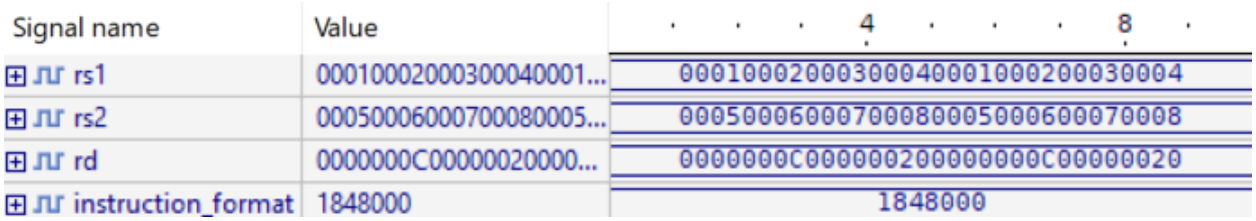


Figure 98. Waveform for test case 1 of MLHU instruction.

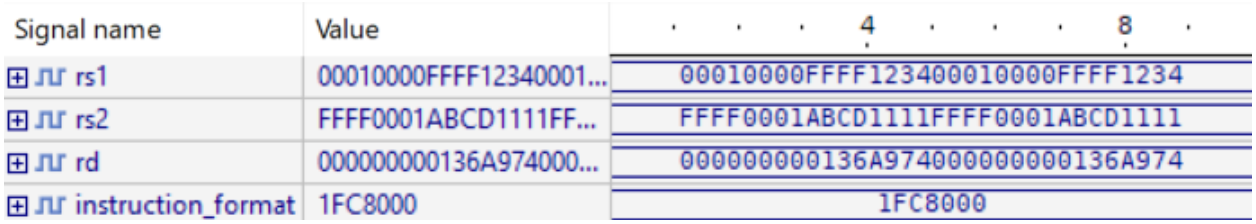


Figure 99. Waveform for test case 2 of MLHU instruction.

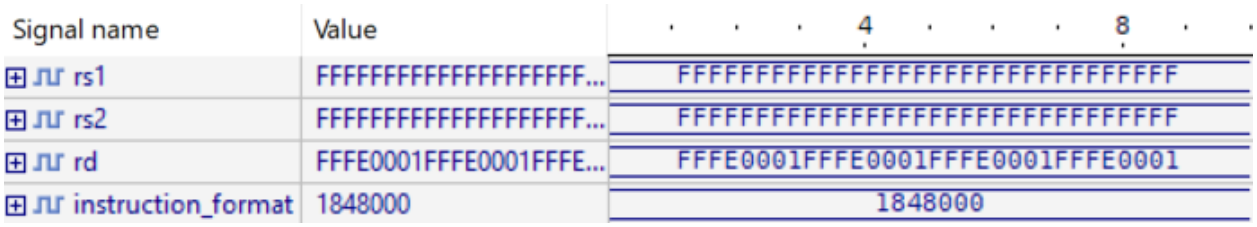


Figure 100. Waveform for test case 3 of MLHU instruction.

Results:

The test cases were designed to test no overflow, mixed bits, and maximum unsigned with overflow values. All test cases were successful and produced expected results.

MLHCU: multiply low by constant unsigned

```
--MLHCU
when "1010" =>
-- store the unsigned 5 bit value of rs2 into 16 bit vector
w0 := resize(unsigned(instruction_format(14 downto 10)), 16);
for i in 0 to 3 loop
    -- store the rightmost 16 bits of rs1 into 16 bit vector
    w3 := unsigned(rs1(i*32 + 15 downto i*32));
    -- multiply the stored 5bit rs2 and 16bit rs1
    product := w0*w3;
    -- store in respective rd
    rd((i*32 + 31) downto (i*32)) <= std_logic_vector(product);
end loop;
```

Figure 101. Source code for MLHCU instruction.

```
-- MLHCU --
report "=== STARTING MLHCU TESTBENCH ===";

-- TEST 1 (mixed bits)
rs1 <= x"00010000FFFF1234000A0005ABCD0102";
rs2 <= x"00000000000000000000000000000000"; -- value is irrelevant so reset
instruction_format <= "11" & "00001010" & "000010100000000";  -- rs2 = 00001 = 1
wait for period;

-- TEST 2 (large constant)
rs1 <= x"00010002000300040005000600070008";
instruction_format <= "11" & "10101010" & "111110000000000"; -- rs2 = 11111 = 31
wait for period;

-- TEST 3 (max halfword * max constant) rs2 = 11111 = 31
rs1 <= x"FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF";
instruction_format <= "11" & "10101010" & "11111"& "00000"& "00000";
wait for period;

report "All mlhcu tests complete!";
```

Figure 102. Testbench code for MLHCU instruction

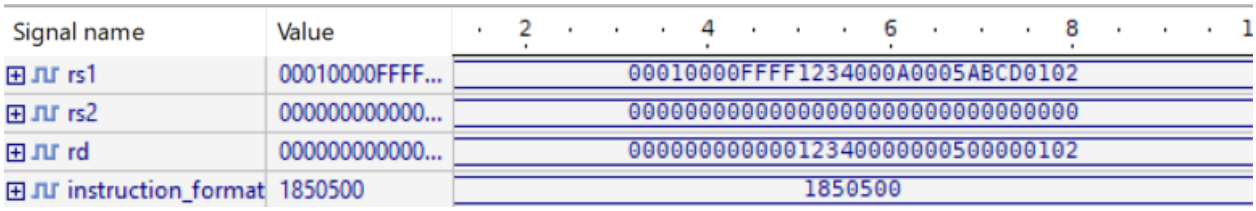


Figure 103. Waveform for test case 1 of MLHU instruction.

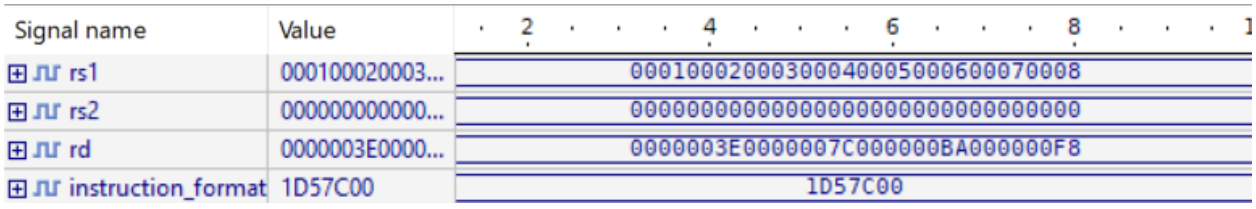


Figure 104. Waveform for test case 2 of MLHCU instruction.

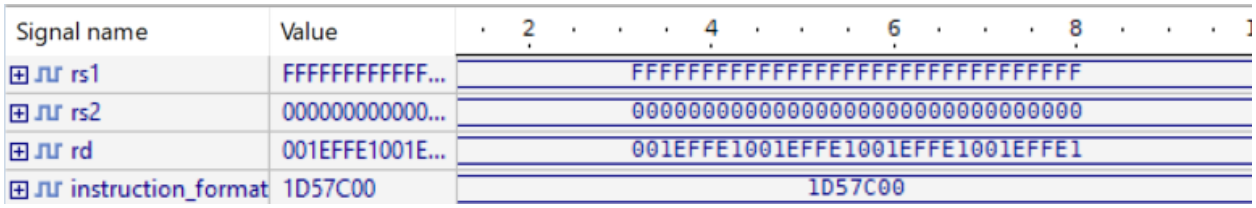


Figure 105. Waveform for test case 3 of MLHCU instruction.

Results:

rs2 is cleared because the value in this register is irrelevant for this operation. Only the five bit value of the instruction field of rs2 matters here, NOT the five least significant bits of the 128-bit value of rs2. We tested cases with rs1 with mixed bits, a large rs2 constant, and multiplying the max halfword with max constant. The test cases were successful and produced expected results.

AND: bitwise logical and

```
--AND
when "1011" =>
rd <= rs1 and rs2;
```

Figure 106. Source code for AND instruction.

```
-- AND --
report "=== STARTING AND TESTBENCH ===";

-- TEST 1
rs1 <= x"0123456789ABCDEF0011223344556677";
rs2 <= x"FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF";
instruction_format <= "11" & "00001011" & "000000000000000";
wait for period;

-- TEST 2
rs1 <= x"FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF";
rs2 <= x"00000000000000000000000000000000";
instruction_format <= "11" & "00001011" & "000000000000000";
wait for period;

report "All and tests complete!";
```

Figure 107. Testbench code for AND instruction.

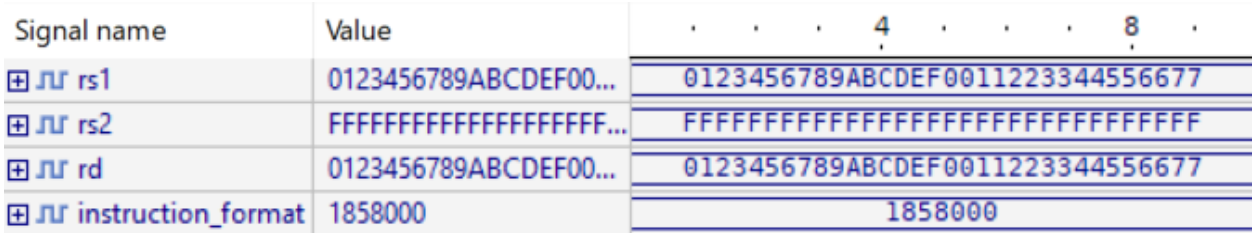


Figure 108. Waveform for test case 1 of AND instruction.

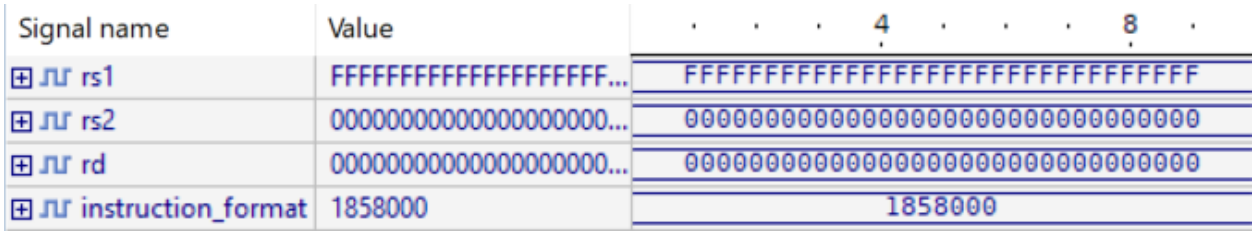


Figure 109. Waveform for test case 2 of AND instruction.

Results:
The test cases were designed to test random values of rs1 with max rs2, and max value of rs1 with min value of rs2. All test cases were successful and produced expected results.

CLZW: count leading zeroes in words

```
--CLZW
when "1100" =>
for i in 0 to 3 loop
    word_val := rs1((i*32 + 31) downto (i*32));

    -- counter for # of zeros
    count := 0;

    -- if all zeros, set count to 32
    if word_val = X"00000000" then
        count := 32;
    else -- else need to look for first 1
        for j in 31 downto 0 loop
            if word_val(j) = '0' then
                count := count + 1;
            else -- it is a 1, then STOP
                exit;
            end if;
        end loop;
    end if;
    rd((i*32 + 31) downto (i*32)) <= std_logic_vector(to_unsigned(count, 32));
end loop;
```

Figure 110. Source code for CLZW instruction.

```
-- CLZW --
report "=== STARTING CLZW TESTBENCH ===";

-- TEST 1
rs1 <= x"8000000080000000000000007FFFFFFF";
rs2 <= x"00000000000000000000000000000000"; -- not needed so reset
instruction_format <= "11" & "00001100" & "000000000000000";
wait for period;

-- TEST 2
rs1 <= x"0F0F0F0FF0F0F0F00FFF0000FF000000";
instruction_format <= "11" & "00001100" & "000000000000000";
wait for period;

report "All clzw tests complete!";
```

Figure 111. Testbench code for CLZW instruction.

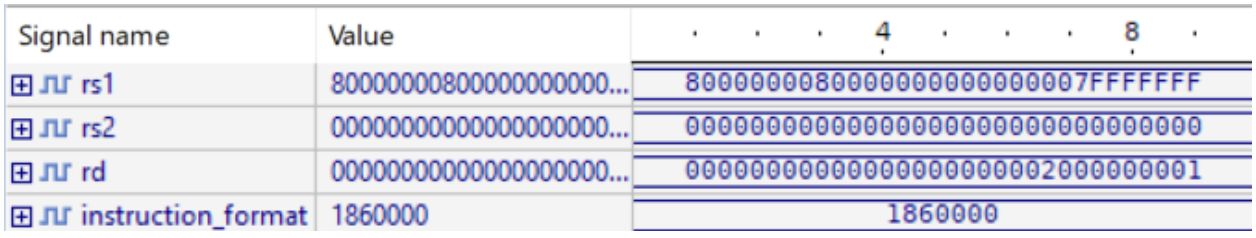


Figure 112. Waveform for test case 1 of CLZW instruction.

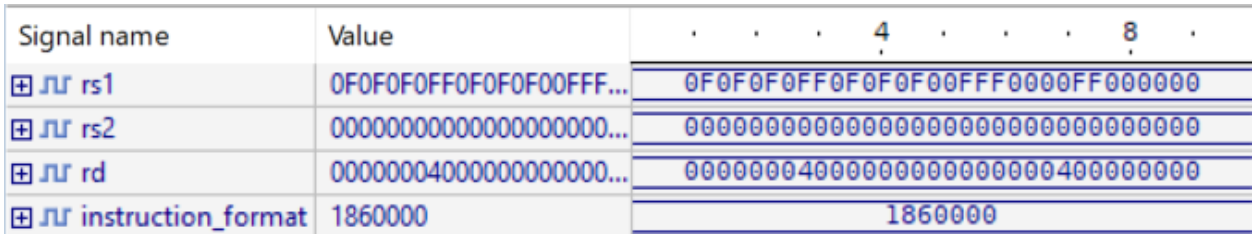


Figure 113. Waveform for test case 2 of CLZW instruction.

Results:

rs2 is cleared because the value in this register is irrelevant for this operation. The test cases were designed with random values for rs1. All test cases were successful and produced expected results.

ROTW: rotate bits in word

```
--ROTW
when "1101" =>
for i in 0 to 3 loop
    rotated_rs1 := rs1((i*32 + 31) downto (i*32)); -- copy 32 orignal into rotated
    num_rot := to_integer(unsigned(rs2((i*32 +4) downto (i*32)))); -- get respective val of 5 bit in each 32 bit

    for j in 1 to num_rot loop
        temp_lsb := rotated_rs1(0); -- save the lsb
        rotated_rs1(30 downto 0) := rotated_rs1(31 downto 1); -- shift right 1 by copying
        rotated_rs1(31) := temp_lsb; -- put lsb into the msb
    end loop;
    rd((i*32 + 31) downto (i*32)) <= rotated_rs1;
end loop;
```

Figure 114. Source code for ROTW instruction.

```
-- ROTW --
report "=== STARTING ROTW TESTBENCH ===";

-- TEST 1 (one-bit rotation)
rs1 <= x"00000001000000020000000480000000";
rs2 <= x"00000001000000010000000100000001";
instruction_format <= "11" & "00001101" & "000000000000000";
wait for period;

-- TEST 2 (max rotation - 31)
rs1 <= x"00000001FFFFFFFFAAAAAAAA55555555";
rs2 <= x"0000001F0000001F0000001F0000001F";
instruction_format <= "11" & "00001101" & "000000000000000";
wait for period;

-- TEST 3 (mixed rotation)
rs1 <= x"F00000000F000000AAAAAAAA55555555";
rs2 <= x"00000004000000080000000C00000010";
instruction_format <= "11" & "00101101" & "000000000000000";
wait for period;

report "All rotw tests complete!";
```

Figure 115. Testbench code for ROTW instruction.

Signal name	Value	4	8
rs1	0000000100000002000...	00000001000000020000000480000000	
rs2	0000000100000001000...	00000001000000010000000100000001	
rd	8000000000000001000...	80000000000000010000000240000000	
instruction_format	1868000	1868000	

Figure 116. Waveform for test case 1 of ROTW instruction.

Signal name	Value	4	8
rs1	00000001FFFFFFFFAAA...	00000001FFFFFFFFAAAAAAAA55555555	
rs2	0000001F0000001F0000...	0000001F0000001F0000001F0000001F	
rd	00000002FFFFFFFF5555...	00000002FFFFFFFF55555555AAAAAAAA	
instruction_format	1868000	1868000	

Figure 117. Waveform for test case 2 of ROTW instruction.

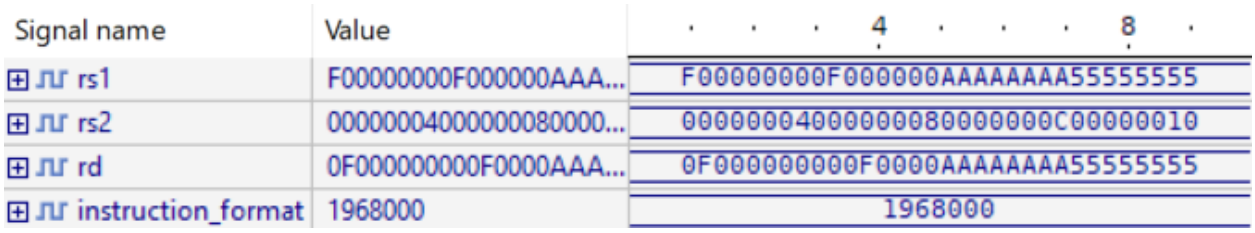


Figure 118. Waveform for test case 3 of ROTW instruction.

Results:

The test cases were designed to test one-bit rotation, max rotation, and random rotation. All test cases were successful and produced expected results.

SFWU: subtract from word unsigned

```
--SFWU
when "1110" =>
for i in 0 to 3 loop
    w1 := unsigned(rs1((i*32 + 31) downto (i*32))); -- store unsigned value of word of rs1
    w2 := unsigned(rs2((i*32 + 31) downto (i*32))); -- store unsigned value of word of rs2
    result := w2 - w1;
    rd((i*32 + 31) downto (i*32)) <= std_logic_vector(result);
end loop;
```

Figure 119. Source code for SFWU instruction.

```
-- SFWU --
report "=== STARTING SFWU TESTBENCH ===";

-- TEST 1 (underflow, borrowing)
rs1 <= x"00000002000000030000000400000005";
rs2 <= x"00000001000000020000000300000004";
instruction_format <= "11" & "00001110" & "000000000000000";
wait for period;

-- TEST 2 (extremes)
rs1 <= x"FFFFFFFF00000000000000000000000";
rs2 <= x"00000000FFFFFFFFFFFFFFFFFFFFFFF";
instruction_format <= "11" & "00001110" & "000000000000000";
wait for period;

-- TEST 3 (alternating bits)
rs1 <= x"AAAAAAAA55555555FFFFFFF00000000";
rs2 <= x"55555555AAAAAAAA00000000FFFFFFF";
instruction_format <= "11" & "00001110" & "000000000000000";
wait for period;

report "All sfwu tests complete!";
```

Figure 120. Testbench code for SFWU instruction.

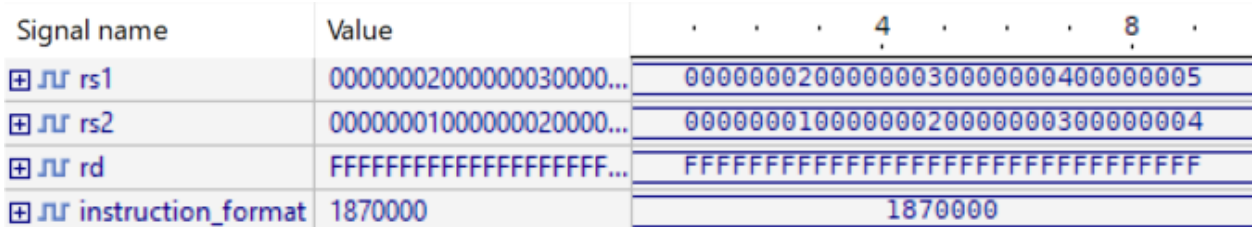


Figure 121. Waveform for test case 1 of SFWU instruction.

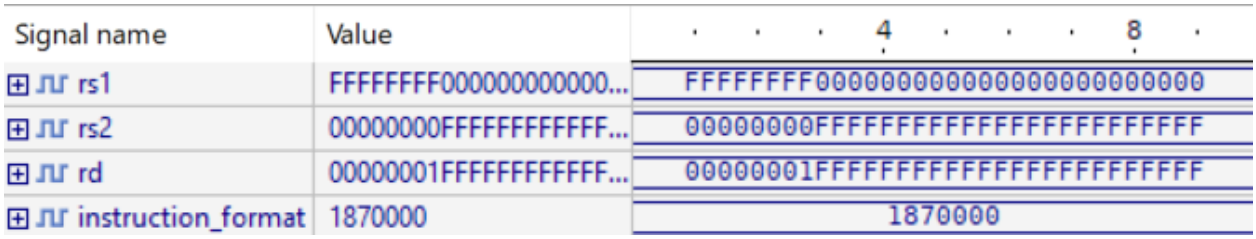


Figure 122. Waveform for test case 2 of SFWU instruction.

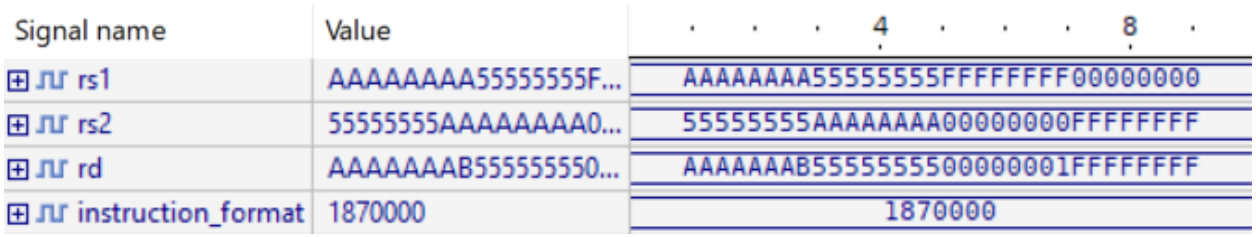


Figure 123. Waveform for test case 3 of SFWU instruction.

Results:

The test cases were designed to test underflow, extreme values, and alternating bits. All test cases were successful and produced expected results.

SFHS: subtract from halfword saturated

```
--SFHS
when "1111" =>
for i in 0 to 7 loop
    reg1 := signed(rs1((i*16 + 15) downto (i*16)));
    reg2 := signed(rs2((i*16 + 15) downto (i*16)));
    diff := resize(reg2, 17) - resize(reg1, 17);

    if diff > to_signed(32767,17) then -- pos overflow, fix
        reg_result((i*16 + 15) downto (i*16)) := to_signed(32767, 16);
    elsif diff < to_signed(-32768,17) then --neg overflow, fix
        reg_result((i*16 + 15) downto (i*16)) := to_signed(-32768, 16);
    else -- value works, keep original
        reg_result((i*16 + 15) downto (i*16)) := resize(diff,16);
    end if;
end loop;
rd <= std_logic_vector(reg_result);
```

Figure 124. Source code for SFHS instruction.

```
-- SFHS --
report "=== STARTING SFHS TESTBENCH ===";

-- TEST 1 (no saturation)
rs1 <= x"0001000200030004000500060007 0008";
rs2 <= x"0008000700060005000400030002 0001";
instruction_format <= "11" & "00001111" & "000000000000000";
wait for period;

-- TEST 2 (+ overflow)
rs1 <= x"8000800080008000800080008000 8000";
rs2 <= x"7FFF7FFF7FFF7FFF7FFF7FFF7FFF 7FFF";
instruction_format <= "11" & "00001111" & "000000000000000";
wait for period;

-- TEST 3 (- overflow)
rs1 <= x"7FFF7FFF7FFF7FFF7FFF7FFF7FFF 7FFF";
rs2 <= x"8000800080008000800080008000 8000";
instruction_format <= "11" & "00001111" & "000000000000000";
wait for period;

-- TEST 4 (mixed signs)
rs1 <= x"7FFF0000800000017FFFFFF80000 0000";
rs2 <= x"00007FFF7FFF8000FFFF00007FFF 8000";
instruction_format <= "11" & "00001111" & "000000000000000";
wait for period;

report "All sfhs tests complete!";
```

Figure 125. Testbench code for SFHS instruction.

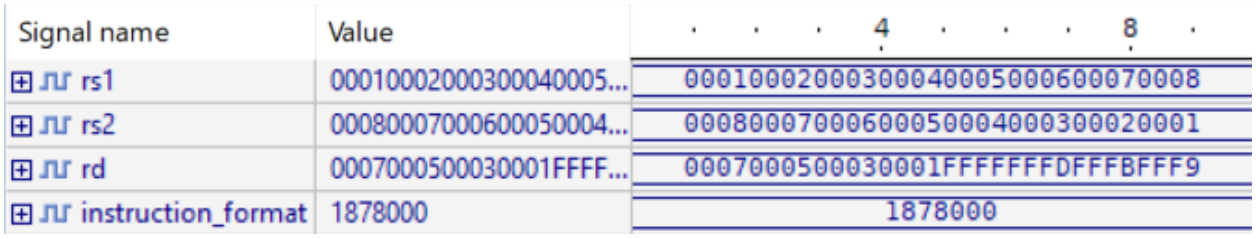


Figure 126. Waveform for test case 1 of SFHS instruction.

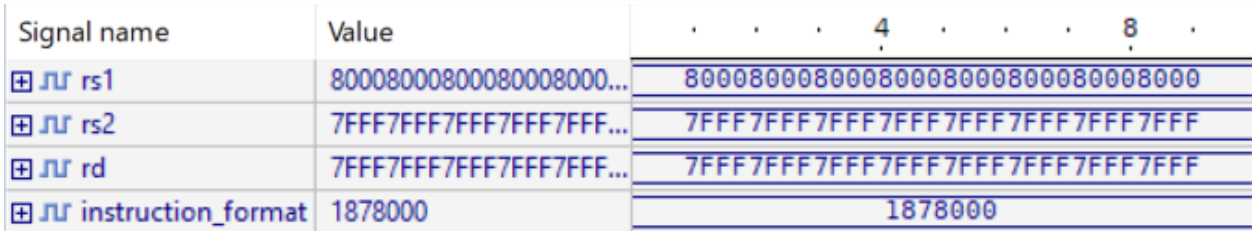


Figure 127. Waveform for test case 2 of SFHS instruction.

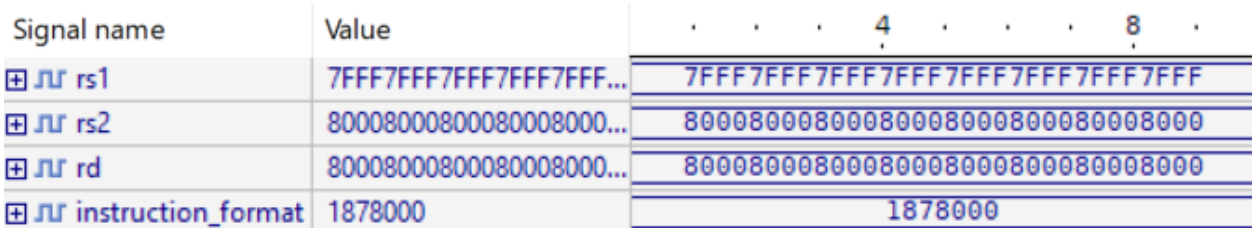


Figure 128. Waveform for test case 3 of SFHS instruction.

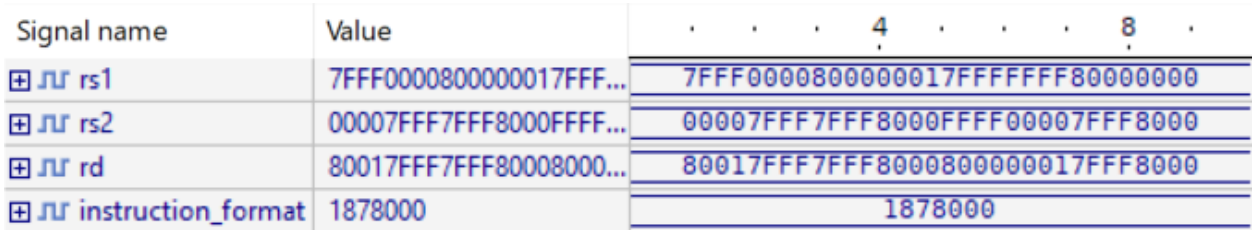


Figure 129. Waveform for test case 4 of SFHS instruction.

Results:

The test cases were designed to test no saturation, positive overflow, negative overflow, and registers with mixed signs. All test cases were successful and produced expected results.

Pipelining Testing

To evaluate pipeline behavior, we constructed a short test program containing four instructions. The goal was to verify correct register initialization, arithmetic execution, and the handling of data hazards between dependent instructions. The program loads values into two registers, performs an arithmetic-unit operation, and then executes a SIMD-style instruction that depends on the previous results.

Human-written instruction sequence:
li r1, 0, 10
li r2, 0, 3
au r3, r1, r2
simal r4, r3, r1, r2
These instructions intentionally introduce read-after-write dependencies, making them useful for observing whether forwarding or stalls occur during execution.

Assembler output (binary):
00000000000000000101000001
00000000000000000001100010
11000000100001000000100011
10000000100000100011100100

Manual hexadecimal conversion:
0x141
0x62
0x1810823
0x1010464
Hexadecimal values were generated manually for easier reference during waveform analysis.

Stage 1: Instruction Fetch (IF)

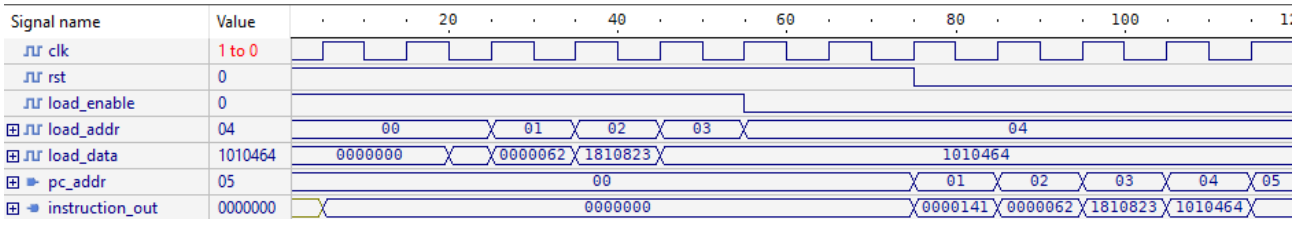


Figure 130. Waveform for instruction fetch.

Verification Signals:
pc_addr: Program Counter value. Increments sequentially only after all instructions are loaded and reset is de-asserted.

instruction_out: Instruction fetched from the Instruction Buffer at the address pointed to by pc_addr. This value then enters the IF/ID pipeline register.
load_enable: Asserted during the loading phase to allow writing into the Instruction Buffer. When high, the instruction on load_data is stored into the location specified by load_addr.
load_addr: Selects the Instruction Buffer entry being written to during program load. This address is incremented to load the instruction sequence.
load_data: 25-bit instruction word written into the Instruction Buffer at load_addr when load_enable is asserted.

Stage 2: Instruction Decode + Register Fetch (ID)

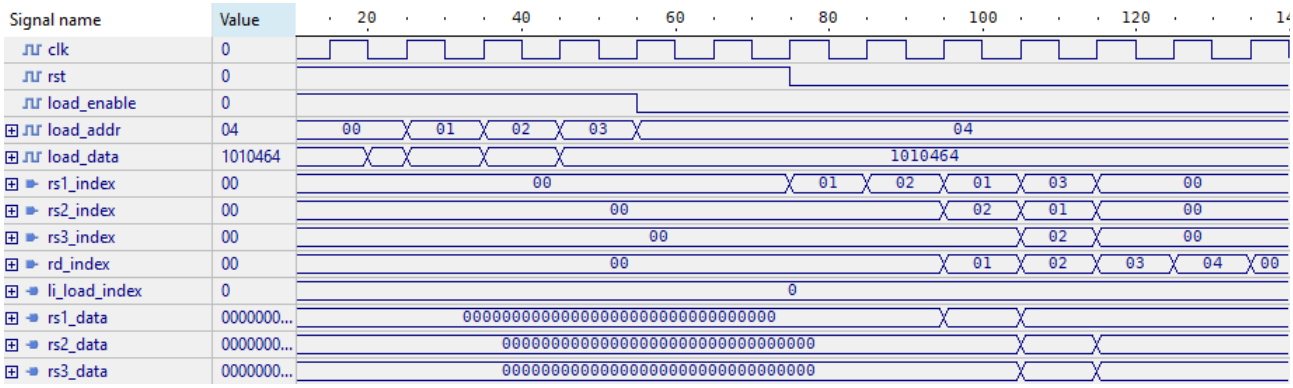


Figure 131. Waveform for decode and register file.

Verification Signals:
rs1_index, rs2_index, rs3_index: Register source indices extracted during Decode. The decoder reads these fields directly from the instruction, and the Register File uses the same indices to output operand values.
rd_index: Destination register index extracted from the instruction. Shared between Decode and the Register File so the pipeline knows which register will receive the result during Writeback.
li_load_index: Only used for li instructions. Extracted from bits 23:21 and used by the Register File to determine which immediate field should be written into rd.
rs1_data, rs2_data, rs3_data: 128-bit register values read from the Register File corresponding to the extracted source indices. Initially zero until the first writeback occurs, confirming correct default initialization.

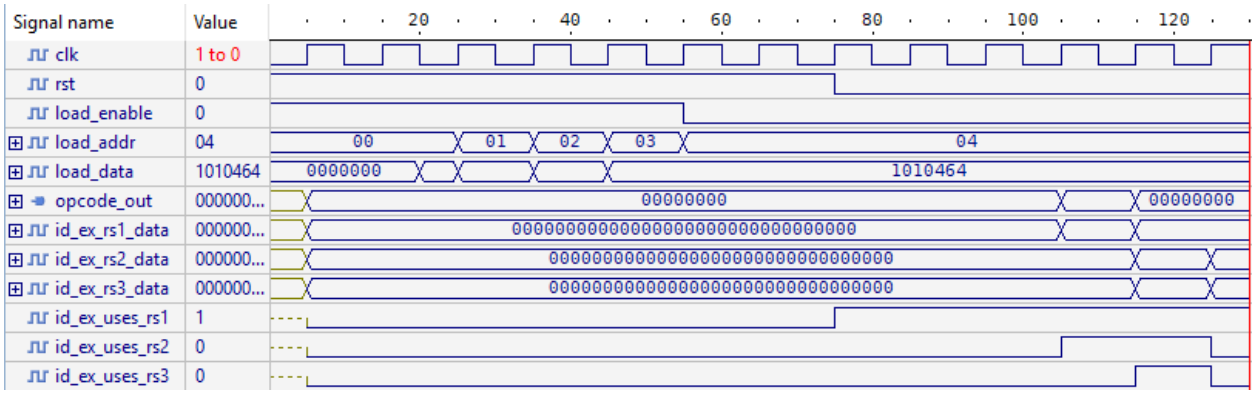


Figure 132. Waveform for ID/EX register.

Verification Signals:

opcode_out: Changes to 00000010 when the au instruction enters ID/EX, then returns to 00000000 when simal progresses through the pipeline. This confirms the opcode is being carried correctly into the Execute stage.

id_ex_rs1_data, id_ex_rs2_data, id_ex_rs3_data: The 128-bit register values read from the Register File during Decode and transferred into the ID/EX pipeline register for use by the Execute stage.

id_ex_uses_rs1, id_ex_uses_rs2, id_ex_uses_rs3: Control signals generated in the Decode stage to indicate whether the instruction uses each source register. These signals are used by the Forwarding Unit to determine if bypassing is required.

Stage 3: Execute

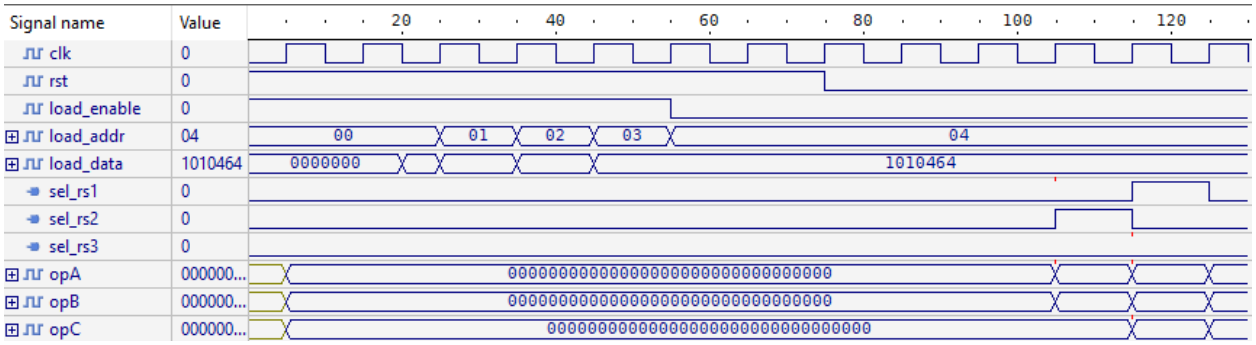


Figure 133. Waveform for data forwarding.

Verification Signals:

sel_rs1, sel_rs2, sel_rs3: These indicate whether rs1, rs2, or rs3 should take the normal ID/EX value or a forwarded value from the EX/WB stage. A value of 1 shows that forwarding is taking place for that operand.

opA, opB, opC: These are the operand values that enter the Multimedia ALU (MALU). Their source depends on the forwarding selection controlled by sel_rs1, sel_rs2, and sel_rs3. As soon

as EX/WB produces a result, opA, opB, and opC update immediately without waiting for a register writeback. This confirms that the forwarding logic is working properly and supplying operands directly when needed.

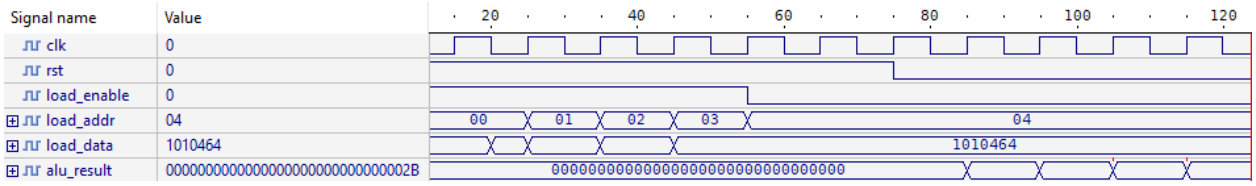


Figure 134. Waveform for MALU.

Verification Signals:

alu_result: Output produced from the MALU after execution. The waveform does not display the numerical result in readable form, but updates occur correctly in response to each instruction, confirming that the ALU is computing expected values.

Stage 4: Write-Back (WB)

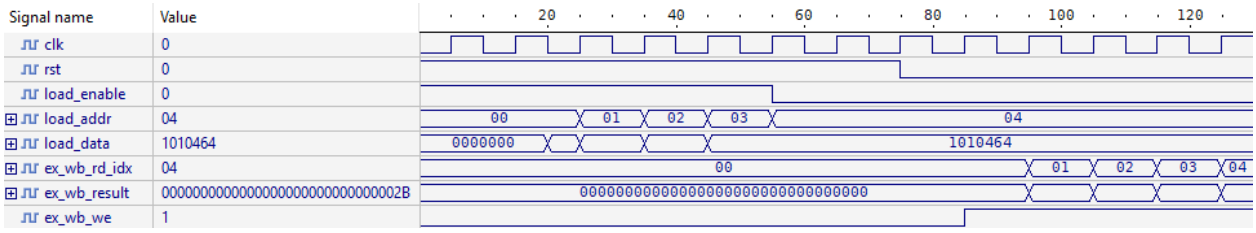


Figure 135. Waveform for EX/WB register.

Verification Signals:

ex_wb_rd_idx: Destination register index stored in the EX/WB pipeline register. This field determines which register will receive the final result during write back.

ex_wb_result: The 128-bit result produced by the MALU and held within the EX/WB register. This value is written to the register file after the write enable signal is asserted.

ex_wb_we: Write enable control for the write-back stage. When this signal is high, the register file updates the register selected by ex_wb_rd_idx with the value in ex_wb_result. When low, no register modification occurs. When ex_wb_we is 1, the register file is updated in the following cycle, confirming correct write-back timing and pipeline completion.

Conclusion

Through the design and implementation of a pipelined 128-bit multimedia processor, we successfully demonstrated the development of a custom CPU with a reduced set of multimedia instructions similar to those found in the Sony Cell SPU and Intel SSE architectures. Beginning with the MALU as the primary execution unit allowed us to verify arithmetic correctness early, which greatly simplified the later integration of pipeline stages. Once data forwarding was implemented, dependent instructions executed without stalls, allowing the processor to maintain continuous throughput.

The creation of a custom assembler and VHDL testbench enabled us to automatically generate programs, load them into the instruction buffer, and verify execution within simulation. Using waveform analysis, we confirmed proper register updates, forwarding behavior, and overall pipeline flow, validating the functionality of the full design.